Model Integration and Transformation
–
A Triple Graph Grammar-based QVT Implementation

Vom Fachbereich 18
Elekrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung der Würde
eines Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

von

Dipl.-Inform. Alexander Königs

geboren in Krefeld-Hüls

Referent:	Prof. Dr. rer. nat. Andy Schürr
Korreferent:	Prof. Dr. rer. nat. Gregor Engels
Tag der Einreichung:	01.07.2008
Tag der mündlichen Prüfung:	31.10.2008

D17
Darmstadt 2009

Danksagung

Die vorliegende Arbeit wäre ohne die Hilfe, Unterstüzung und den Beistand einer ganzen Reihe von Leuten nicht möglich gewesen. Daher möchte ich mich herzlich bei Allen bedanken, die auf die ein oder andere Art und Weise direkt oder indirekt zu dieser Arbeit beigetragen haben.
Im Einzelnen möchte ich besonders folgenden Personen danken:
Meinem Doktorvater Professor Andy Schürr für die Chance zur Promotion, die beispiellose Betreuung meiner Arbeit, zahlreiche ergiebige Diskussionen, wertvolle Anregungen und seine große Geduld. Für seine Tätigkeit als erster Gutachter und Prüfer und nicht zuletzt als großartiger Chef.
Professor Gregor Engels für seine Bereitschaft, die Rollen des zweiten Gutachters und Prüfers zu übernehmen.
Den Professoren Ralf Steinmetz und Jürgen Adamy als dritte bzw. vierte Prüfer.
Tobias Rötschke und Johannes Jakob nicht nur als Kollegen, sondern vor allem auch als Freunde für zahlreiche, zum Teil leidenschaftlich geführte Diskussionen, gegenseitige Motivation und Freizeit in Form von Pool Billard, Poker und Tischfußball.
Felix Klar für seine unermüdliche Unterstützung bei der Realisierung unseres TGG-Ansatzes und dessen Fortführung.
Carsten Amelunxen, Markus Schmidt und Oliver Alt als geschätzte Kollegen der ersten Stunde.
Elodie Legros, Ingo Weisemöller und Patrick Mukherrje als die zweite Generation von wissenschaftlichen Mitarbeitern am Fachgebiet.
Den zahlreichen Studenten, vor allem Emre Karaca, die durch ihre Studien-, Bachelor-, Diplom- und Master-Arbeiten und ihre Tätigkeiten als studentische Hilfskräfte die Grundlage für die vorliegende Dissertation geschaffen haben.
Unserem technischen Administrator Ingo Heip für seinen beispiellosen Einsatz, eine ausgezeichnete technische Infrastruktur bereitzustellen, und das Einweihen in die Geheimnisse des Geocachings.
Meinen Freunden und nicht zuletzt meiner Mutter und meiner Großmutter für die moralische Unterstützung und Motivation, ohne die diese Arbeit nicht möglich gewesen wäre.

Contents

List of Figures

Abstract

Nowadays, software and system development projects involve an increasing number of various CASE tools each of which is specialized in certain tasks or phases of the development process. This results in an unrelated distribution of the data of a project as a whole over the different data repositories of the considered tools. The task of manually keeping the data consistent is cumbersome, time consuming, and error prone. Therefore, there is an urgent need for automatic support in data consistency checking and consistency enforcement. OMG's Query / View / Transformation (QVT) standard provides a model-based language for the specification of consistency checking and consistency enforcement rules. The QVT standard currently is implemented by a number of different groups but suffers from the fact that it lacks a proper formalization up to now. In contrast Triple Graph Grammars (TGGs) provide a declarative language for the specification of consistency checking and consistency enforcement rules based on the formal foundation of graph grammars. However, TGGs lack some concepts provided by the QVT standard which are needed in practice to be applicable. This work transfers TGGs into OMG's world of metamodeling and extends them by the desired concepts from QVT. The result is an TGG-based implementation of the QVT standard based on the formalism of graph grammars. Furthermore, the presented approach will be supplemented by a framework for automatically checking and enforcing the consistency of distributed data of a considered development project as a whole.

Zusammenfassung

In der Software- und Systementwicklung kommen immer häufiger auf bestimmte Aufgaben oder Phasen des Entwicklungsprozess zugeschnittene Werkzeuge zum Einsatz. Daraus resultiert eine lose Verteilung der Projektdaten über voneinander unabhängige Datenspeicher. Die erforderliche Konsistenzhaltung der Daten ist auf manuellem Weg sehr kostenintensiv und fehleranfällig. Deshalb ist eine automatische Unterstützung zur Konsistenzprüfung und Konsistenzwiederherstellung wünschenswert. Der Query / View / Transformation (QVT) Standard der OMG definiert eine modellbasierte Sprache zur Spezifikation von Regeln zur angestrebten Prüfung und Wiederherstellung der Konsistenz von Daten. Derzeit arbeiten zahlreiche Gruppen an Implementierungen dieses Standards, dem es aber bis heute an einer formalen Grundlage fehlt. Tripel-Graph-Grammatiken (TGGen) hingegen bieten eine deklarative Sprache zur Spezifikation von Konsistenzprüfungs- und Konsistenzwiederherstellungsregeln auf der formalen Grundlage von Graphgrammatiken. TGGen fehlt es allerdings an wichtigen in der Praxis benötigten Konzepten, die der QVT-Standard bietet. Diese Arbeit überträgt den Ansatz der TGGen auf die Metamodellierungswelt der OMG und erweitert ihn um fehlende Konzepte des QVT-Standards. Ziel ist eine eigene Implementierung des QVT-Standards, deren Semantik sich auf den bestehenden Formalismus der Graphgrammatiken stützt, angewandt von einem Rahmenwerk zur automatischen Unterstützung zur Konsistenzhaltung und Konsistenzwiederherstellung von über voneinander unabängigen Datenspeichern verteilten Projektdaten.

1 Introduction

1.1 Motivation

Current software and system development process models subdivide the flow of all involved activities into several phases or tasks (c.f. Fig. 1.1). For instance the waterfall model [Som06] from Fig. 1.1a as a very simple and basic process model introduces phases as requirements specification, design, construction, integration, testing, installation, and maintenance.

The V-model [Som06] as presented in Fig. 1.1b contains a specification part which includes tasks as user requirements, functional, and design specifications. These specifications are then validated by a testing phase which includes tasks as installation, operational, and performance qualification.

Last but not least the Rational Unified Process (RUP) [Som06] from Fig. 1.1c defines phases called inception, elaboration, construction, and transition. Each phase involves tasks as business modeling, requirements elicitation, analysis and design, implementation, testing, deployment, configuration, and project and process management.

In practice most development teams use their own set of Computer Aided Software Engineering (CASE) tools each of which is specialized in a number of the presented phases or tasks. Fig. 1.2 presents an example of such a tool chain. Usually, requirements elicitation is performed using tools such as Doors[1], Microsoft Word[2], and so on. The modeling of the structural and behavioral parts of the considered system can be done by using tools such as Enterprise Architect[3], Matlab (Simulink / Stateflow)[4], Rational Rose[5], and so on. In case of an embedded system, the hardware design usually is done by applying tools that support

[1] http://www.telelogic.com/products/doors/
[2] http://www.microsoft.com/word/
[3] http://www.sparxsystems.com/
[4] http://www.mathworks.com/products/matlab/
[5] http://www.ibm.com/software/awdtools/developer/rose/

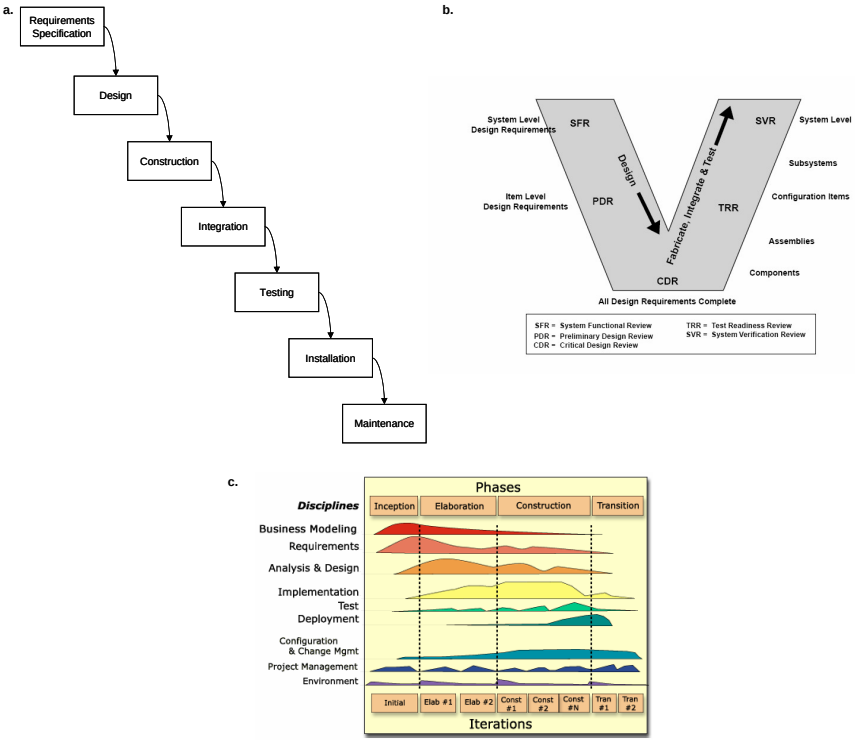

Figure 1.1: Well-known process models

the hardware description language (HDL) as HDL Author[6] for instance and then modeling the desired system using Computer Aided Design (CAD) tools such as Catia[7]. The desired test suites for testing the system under development can be specified by tools such as CTE[8]. Finally, tools such as Windchill[9] manage common products' data.

[6] http://www.mentor.com/products/fpga_pld/hdl_design/hdl_author/
[7] http://www.3ds.com/products/catia/catia-discovery/
[8] http://www.systematic-testing.com/
[9] http://www.ptc.com/products/windchill/

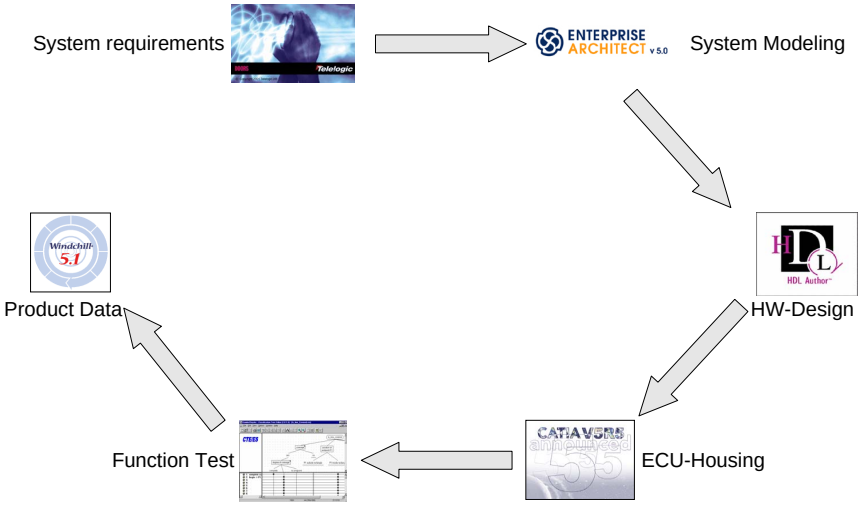

Figure 1.2: Example of a tool chain in a system development process

As a result the data of the development project as a whole is distributed between the different data repositories of the involved tools. Typically, the involved tools are commercials off the shelf (COTS) from different vendors and are seldom integrated with each other. Thus, the relationships and dependencies of the distributed data are invisible to the tools' users. In a typical project there can be many ten-thousands of data dependencies at fine-grained level which must be checked and maintained in order to keep the data of the project as a whole consistent. Performing these tasks manually is cumbersome, time-consuming, error prone, and in the end a nightmare. Moreover, while system development process execution these tools are concurrently used by up to hundreds of developers. Inescapably, projects' data cannot be kept in a consistent state without at least semi-automatic support. Current standards (e.g. IEC 61508[10]) insistently demand a certain level of consistency and data traceability especially for safety-critical systems. Therefore, there is an urgent need for solutions that deal with the integration of development tools and their corresponding data repositories.

[10] http://www.iec.ch/61508/

1.2 Scope

Concerning tool integration Brown [BCM+94] identifies three dimensions of integration: Presentation, Control, and Data integration. Basically, presentation integration aims at integrating tools by providing a uniform user interface for the considered tools. Either this can be achieved by a tool suite which tools rely on more or less the same user interface or by using an integrated tool environment such as the Eclipse platform for instance. Presentation integration is ineligible when integrating COTS tools since the user interfaces of such tools are more or less immutable. Furthermore, presentation integration does not solve the problem of data consistency presented above. Therefore, presentation integration is out of scope for this work. Control integration aims at keeping the data of different tools continuously consistent by notifying all tools about changes that occur in each tool. Control integration is not applicable to our scenario because it requires some sort of event notification mechanism that is seldom provided by COTS tools. Besides, continuously modifying the data of the to be integrated tools is not satisfactory. As mentioned before the different tools should be concurrently usable by hundreds of developers. On the one hand it seems to be a hard task to efficiently handle the number of change events that might occur in a concurrent scenario with many users. On the other hand it would be undesirable that the data a single developer is working on would continuously change due to changes made by different users in different tools. Therefore, control integration is not suitable for our intended scenario and, thus, is out of scope for this work. In this work we put the focus on data integration. To this end we have to address the issues Data Persistence and Data Semantics according to [BCM+94]. Dealing with data persistence means answering the question which data must be kept persistent and where. Furthermore, we must clarify in which way and to which degree data is shared by the to be integrated tools. Data semantics means specifying types of dependencies between data of the considered tools and maintaining information on actual dependencies at runtime.

In addition to the classification presented so far [BCM+94] subdivides integration approaches into a-priori and a-posteriori integration approaches. A-priori integration approaches develop tools that are designed to easily integrate with each other. A-posteriori integration approaches aim at integrating already existing tools regardless whether or not they have been developed having integration purposes in mind. Since the latter usually applies to COTS tools we focus on a-posteriori tool integration in this work.

1.3 Case studies

Traceability link creation

In our first case study our industrial partner DaimlerChrysler is facing the problem to analyze which data objects in one tool correspond to which data objects in another tool. In particular system requirements for a windscreen wiper with a rain sensor are stored in the requirements elicitation tool Doors[11]. Furthermore, corresponding test case specifications are stored in a tool called CTE[12] which supports the classification tree method for black-box testing purposes. In order to specify which requirement is tested by a test case the number of the requirement is stored in the comment field of the regarded test case. Thus, a user is able to manually check whether all requirements are checked by at least one test case. Since there normally are hundreds of requirements for a regarded system this approach is cumbersome, time consuming, and error prone. The situation becomes even worse if the user wants to determine all test cases that are assigned to a regarded requirement. To address the latter issue DaimlerChrysler together with the TFH Berlin, University of Paderborn, and TU Darmstadt developed the Toolnet Framework [ADS02] which enables its users to manually create traceability links between objects stored in different tools. Provided that all valid traceability links have been created beforehand it is very easy to check whether or not all requirements are tested and which test cases are assigned to each requirement. However, Toolnet does not provide any support for automatically creating and validating the needed traceability links. Therefore, our approach should offer such a support.

Model-Model consistency analysis

In another case study Philips Medical Systems is developing magnetic resonance tomographs [Röt09]. Philips is confronted with the situation that the developers initially started the development without a proper architecture of the systems in mind. In the meantime the systems have evolved over many generations and Philips is running into maintenance problems. To address these problems Philips has specified a supposed system architecture and aims at modifying their existing systems in such a way that they meet the envisioned architecture one day. To this end Philips maps the existing source code of the regarded systems to the new architectural concepts. After that Philips compares the current architectures

[11] http://www.telelogic.com/products/doors/
[12] http://www.systematic-testing.com/

with the desired ones and calculates a number of metrics. Thereby, one current architecture constitutes one model, whereas one desired architecture constitutes the other model. Since a metric is seldom useful as it is Philips rather keeps track of the evolution of the calculated metrics over time. This allows for a more sophisticated evaluation of whether or nor the architectures of the maintained systems are converging with the desired architectures.

Model-Model transformation

In another case study our industrial partner from Bosch has to test their embedded automotive multimedia systems [Alt07]. Bosch wants to have support for specifying test cases on an abstract level that considers categories of multimedia systems (e.g. CD player, navigation system) rather than concrete products. In order to test a concrete product Bosch wants to automatically transform the corresponding abstract test case specification into a test case for the considered product. Therefore, Bosch needs support for specifying the transformation of an abstract test case into a corresponding concrete test case.

Running example

As a final case study we examine the task of automatically transforming a given class diagram into a corresponding database schema. We use this task as a running example throughout this work for the following reasons. First of all, the example is rather small and easy to understand. Nevertheless, the transformation of class diagrams into database schemas has proved to be more challenging than most examples with an industrial background. Therefore, this case study allows us to introduce and explain the more sophisticated concepts of our approach in detail. Furthermore, this example has been part of the Model Transformation in Practice Workshop 2005[13]. The participants of this workshop have been asked to tackle this transformation task using their own approaches. Finally, a solution of this transformation task is included in the final QVT specification which is an upcoming model integration standard as introduced in detail in Chapter 3. Thus, this case study can be considered as an official benchmark for QVT-related model transformation approaches. We describe this case study in detail in Section 3.1.

[13] http://sosym.dcs.kcl.ac.uk/events/mtip05/

1.4 Contributions

Having the mentioned case studies in mind we ultimately are aiming at an integrated approach that is able to identify and maintain traceability information between two models, checks two models for consistency, and bidirectionally transform one model into another and vice versa.

To this end we investigate two already existing approaches that claim to be able to cope with the intended model integration tasks. On the one hand we examine the upcoming model integration standard QVT [OMG05b] from the OMG[14]. On the other hand we examine the graph grammar-based approach of TGGs. As we will see later both approaches have their own edges and flaws. Therefore, we aim at combining both approaches in order to compensate the flaws of the one approach by the edges of the other approach and vice versa.

Particularly, we start with the initial TGG approach as presented by Schürr in 1994 [Sch94]. This approach relies on the idea of declaratively specifying model integration rules from which operational rules can be derived automatically. The resulting operational rules can be applied for model integration tasks such as creating and validating traceability links between elements of the to be integrated models, forward and backward model transformation where one model is created from the other and vice versa. We extend this set of classical model integration rules by rules for deleting traceability links and bidirectionally propagating attribute value changes as well as the deletion of model elements.

Furthermore, we come up with strategies of how to apply the derived operational rules automatically for realizing automatic model integration support. To this end we extend the initial TGG approach with the concept of priorities which allows for the sophisticated resolution / avoidance of rule application conflicts. By adding parameters to the declaration of model integration rules we clarify the concept of specifying attribute value expressions and their processing at rule derivation time. Moreover, we investigate the contended concept of Negative Application Conditions and decide to intentionally exclude them from our approach as they can often be simulated with the concept of priorities.

Finally, we adopt some very useful and user-friendly concepts from MOF and QVT. For instance, we adopt MOF's concepts for modularization and reusability. Furthermore, we adopt the concept for explicitly controlling the rule application

[14] http://www.omg.org

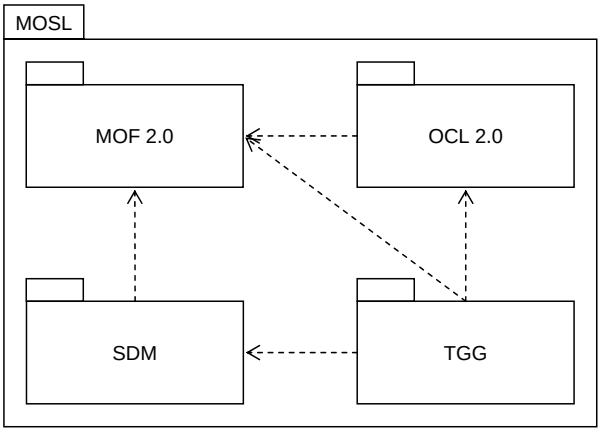

Figure 1.3: Composition of the MOFLON Specification Language (MOSL)

order from QVT. As a result we come up with a model integration approach that the reader hopefully considers to be highly expressive, user-friendly, and formally well-defined.

1.5 Overall picture

This thesis presents work that has been done as part of the research activities of the Real-Time Systems Lab, TU Darmstadt, Germany. The general goal of our lab is to provide a meta-CASE (Computer Aided Software Engineering) tool. A meta-CASE tool is a CASE tool which can be used to specify and implement CASE tools. The tasks of the lab are to provide an appropriate and easy to use specification language, implement a tool that allows for the application of this language and generate corresponding code, and apply our approach to case studies that demonstrate the usefulness of our approach.

As we will discuss later on in detail we have chosen to realize the desired specification language on the foundation of the Meta Object Facility (MOF) [OMG06a] and related standards as proposed by the Object Management Group (OMG) and the formalism of (Triple) Graph Grammars [Sch94]. As illustrated in Figure 1.3

our envisioned specification language is subdivided into four sublanguages (i.e., MOF [OMG06a], OCL [OMG06b], SDM [Zün01], and TGGs [Sch94]). Although we introduce each of these sublanguages throughout this work, this work focuses on the specification and realization of the TGG part.

In order to keep things simple we disregard the details of the integration of the TGG sublanguage with the other sublanguages throughout this thesis. However, the reader is advised to keep in mind that in fact all sublanguages are properly integrated with each other as presented in [Ame08].

1.6 Outline

This work is structured as follows. Chapter 2 introduces OMG's world of metamodeling in general, whereas Chapter 3 introduces the QVT standard in detail. Graph grammars as the formal foundation of our approach are then presented in Chapter 4. In Chapter 5 we introduce a language that allows for the specification of types of data dependencies. This language is then complemented by a declarative rule-based language in Chapter 6 that allows for the specification of rules that can be applied to establish concrete data dependencies at runtime, check and enforce them if they are violated later on. We explain how we automatically translate these declarative rules into operational rules that can be applied in order to support the desired use cases in Chapter 7. Chapter 8 shows the realization of our approach as part of the MOFLON tool suite[15]. We then demonstrate the application of our realized approach to some of our case studies in Chapter 9. In Chapter 10 we compare our approach with similar ones. Finally, Chapter 11 summarizes our results and discusses open issues as well as future work.

[15] http://www.moflon.org

2 Metamodeling

In this chapter we introduce OMG's[1] world of metamodeling. The OMG characterizes itself on its website[2] as an international open membership, not-for-profit computer industry consortium that develops modeling (e.g. MOF, UML), integration (e.g. QVT), and middleware (e.g. CORBA) standards. As the standards specified by the OMG are widely accepted by industry and research we aim at founding our own approach on top of these standards where possible and reasonable.

We start with presenting OMG's vision of software and system engineering according to the Model Driven Architecture (MDA) approach. After that we introduce the Meta Object Facility (MOF) that basically allows for the visual specification of the syntax and static semantics of modeling languages. Finally, we briefly describe the Object Constraint Language (OCL) that complements the MOF by textual constraints for clarifying the static semantics of the considered modeling languages.

2.1 MDA

Model Driven Architecture (MDA) [OMG03] is OMG's vision of software and system development. MDA addresses the problem that arises in traditional software development projects. Documents such as requirement documents, system architectures, UML diagrams created in the early phases of a considered development project usually are only regarded and maintained until coding starts. Then, the development process is mainly focused on manually implementing the desired system. Changes to the system are usually done directly at implementation level and are seldom propagated back to requirements and design documents. This works quite well as long as the development team does not change. Nevertheless,

[1] Object Management Group
[2] http://www.omg.org

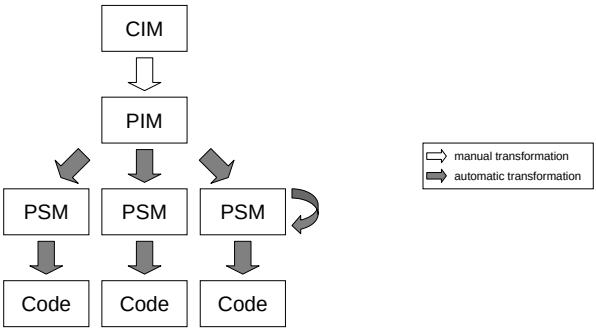

Figure 2.1: The MDA approach

this approach results in an increasing number of inconsistencies between documents from early stages of the project and their actual realization. This exacerbates new developers which might be consigned with maintaining the developed system later on from understanding the system at all. It is hardly possible to understand a foreign system just by aimlessly looking at the code. Rather, new developers need a consistent documentation that presents the system at different levels of abstraction and from different points of view.

MDA envisions to shift the work of the developers to higher levels of abstraction and then generating the source code of the desired system automatically by applying multiple transformation steps. Changes to the considered system should only be necessary and allowed at high level documents from which the transformation to the desired system starts. Therefore, the abstract documents are inherently consistent with the realized system.

According to MDA developers have to perform the following steps in order to realize a system (c.f. Figure 2.1). First of all, the developers must analyze which business processes should be addressed. At this stage developers should neglect any computational issues. Hence, the developers are designing a *Computational Independent Model* (CIM). After that, the developers manually transform this CIM into a more concrete model that takes computational issues into account but still disregards platform-specific details (e.g. desired programming language). Hence, the developers are designing a *Platform Independent Model* (PIM). From this PIM the developers can automatically derive a number of *Platform Specific*

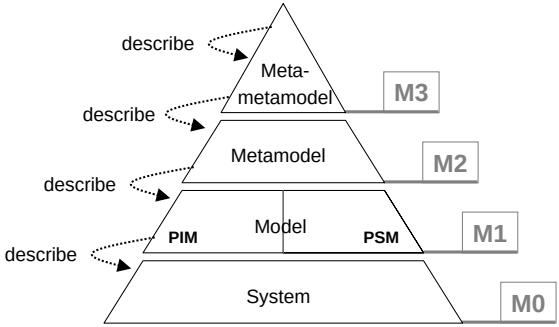

Figure 2.2: OMG's modeling layers

Models (PSM) by applying model transformations. The needed model transformations must have been developed beforehand. The vision is that these transformations are independent from the concrete system development project and can be applied in other projects as well. Finally, the applied transformations result in the desired system. Any changes to this system may only be done at PIM level but not at any PSM level or the resulting code.

2.2 MOF

In order to perform model transformations we must come up with a definition of the term *model*. In the literature there still are ongoing debates on this definition. Throughout this work we adopt the definition given in [OMG03]. Generally speaking a model is an abstraction of something that exists in reality. Abstraction means that a model omits details of the to be modeled entity that are not important from the viewpoint of the model designer. In order to perform computations (e.g. analysis, transformations) on a model the model must be written in a well-defined language. A well-defined language has well-defined syntax and semantics. Furthermore, the language must be suitable for automated interpretation by a computer. *Model transformation* means that we take one model written in a well-defined language as input and produce a new model written in the same or a completely different well-defined language as output.

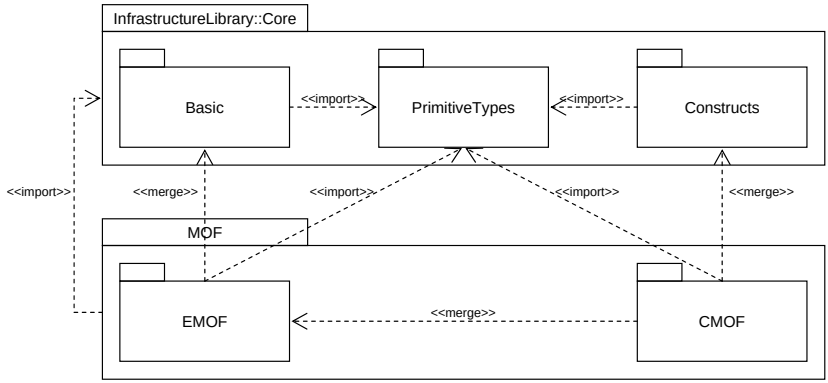

Figure 2.3: Package structure of MOF

The concept for specifying a well-defined language which can be used to describe models is called *metamodeling*. A metamodel itself is a model written in a well-defined language. The metamodel describes the language which in turn describes the desired models. Figure 2.2 illustrates the modeling layer architecture as proposed by the OMG. At the lowest layer M0 (instance layer) resides the to be modeled system. The PIM and PSMs which model the system according to the MDA approach at different levels of abstraction belong to the next higher layer M1 (model layer). Nowadays, these models are often expressed in the various diagram types provided by the Unified Modeling Language (UML). The UML provides diagram types for use case driven requirements engineering (e.g. Use Case diagrams), diagram types for the static structure of a system (e.g. class diagrams), diagram types for expressing the behavior of a system (e.g. state charts, sequence diagrams), and so on. As stated above modeling languages themselves are models which are defined at layer M2 (metamodel layer) of OMG's layer architecture. Again, languages that can be used to define modeling languages such as UML are models of the next higher layer M3 (metametamodel layer). At this layer the OMG has defined the Meta Object Facility (MOF). In order to avoid further recursion in the layer architecture the OMG designed the MOF using itself as its modeling language. The semantics of the MOF is given in plain text [OMG06a]. Furthermore, the MOF coincides with a restricted version of UML class diagrams. Therefore, the OMG factored out the common part of MOF and

UML class diagrams into the UML infrastructure library [OMG07] which is then reused and adapted in order to define MOF and UML class diagrams. Figure 2.3 clarifies the relationship between MOF and the UML infrastructure library. The UML infrastructure library contains a package `Core` which in turn contains the packages `Basic`, `PrimitiveTypes`, and `Constructs`. The `Core` package is imported by the `MOF` package in order to access the contained packages. The `MOF` package itself contains two packages `EMOF` and `CMOF`. The package `EMOF` package contains a subset of MOF called *Essential MOF* (EMOF). EMOF is designed to correspond closely to facilities found in common object-oriented languages in order to allow easy tool development and integration. In contrast the package `CMOF` contains the *Complete MOF* (CMOF). CMOF provides more sophisticated concepts than EMOF and is used to specify modeling languages such as UML. The package `CMOF` merges the packages `EMOF` and `Core`. In contrast to an import which only allows to access elements contained in the imported package a *merge* means the following[3]. Extending a given package just by using imports requires the user to manually create a new class in the new package for each class in the given package. Thereby, the user has to manually ensure that each class in the new package inherits from the corresponding class of the given package. After that the user can add their intended extensions to the new package. In contrast the package merge relieves the user of the presented manual steps. Conceptually, for each element in the merged package the merge creates a corresponding element in the merging package and relates them by a generalization. If the merging package declares a class which name matches the name of a merged class the declared class just inherits from the original class. Therefore, the package `CMOF` contains all concepts from the `EMOF` and the `Core` package. From now on every time we talk about MOF we actually refer to CMOF.

The most important part of the metamodel of the (C)MOF is shown in Figure 2.4. Basically, a MOF model consists of classes that can be related with each other using associations. Furthermore, classes contain operations and properties. Properties can be either attributes or association ends. To put it simple classes can be regarded as types of model elements, whereas associations can be seen as relations (i.e., typed sets of links) between model elements. In contrast to the former MOF 1.4 standard the current version 2.0 introduces more sophisticated concepts for modularization and generalization as depicted in Figure 2.5. Basi-

[3] Technically we rely on an out-dated semantics of merge. We are the opinion that the old semantics is more useful and intended compared to the actual semantics described in the latest version of the standard.

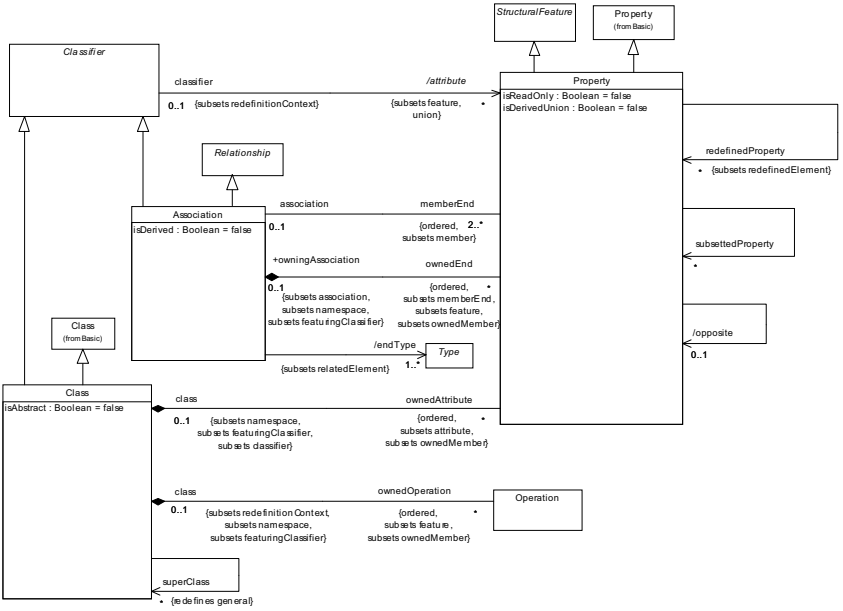

Figure 2.4: Cut-out of MOF's metamodel taken from [OMG06a]

cally, a package contains packageable elements (e.g. classes, associations, and packages). Since packages are namespaces they can import each other. Furthermore, packages can be related to each other by package merges as introduced above. For the complete MOF 2.0 metamodel and its semantics the reader is referred to [OMG06a]. For a detailed comparison between MOF 1.4 and MOF 2.0 the reader is referred to [Ame08].

Figure 2.6 gives an example for the application of OMG's layered modeling architecture. At instance layer M0 there is an picture of the to be modeled system (i.e., an integrated circuit (IC)). At modeling layer M1 we see the logic symbol of an AND gate at the left hand side and a schematic representation of the package of the IC at the right hand side. Both elements can be used to model a certain aspect of the system. The definition of symbols and packages resides on the

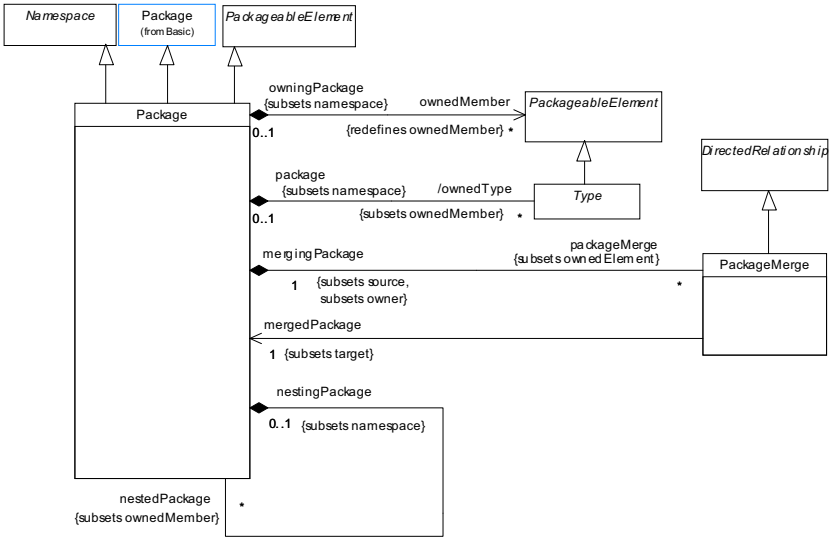

Figure 2.5: Package concepts of MOF taken from [OMG07]

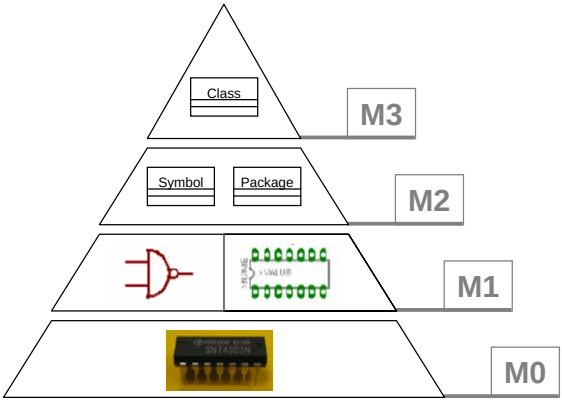

Figure 2.6: Exemplary application of OMG's layered modeling architecture

Figure 2.7: Exemplary metamodel **a.** without and **b.** with an OCL constraint

metamodeling layer M2. Both definitions rely on the concept of classes that is defined at the metametamodeling layer M3 as provided by the MOF.

2.3 OCL

As stated above MOF can be used to write a metamodel that describes which structural requirements a model must satisfy in order to conform to the metamodel. Typically, there are further constraints a model should adhere to which cannot reasonably be expressed using a graphical notation. In the metamodel from Figure 2.7a. there is a class `Person` associated with a class (Driving-)`License`. Without further constraints every model that links a `Person` regardless of its age with an arbitrary number of `Licenses` conforms to the metamodel. Typically, a `Person` can only obtain a `License` if its age is at least 18 for instance. MOF itself provides no means for expressing this essential constraint at all. Therefore, the OMG complemented the MOF with the textual object constraint language (OCL) [OMG06b]. A typical OCL constraint is shown in Figure 2.7b. The constraint states that all `Persons` that are associated with at least one `License` must be at least 18 years old. To this end the constraint is evaluated in the given context of class `License`. For each `License` the owner is determined. Finally, the constraint tests whether the age of each owner is at least 18. If there is at least one `Person` whose age is less than 18 that owns a `License` the constraint evaluates to `false` otherwise to `true`. By design the OCL can only be evaluated without any side effects, i.e. a constraint cannot modify a given model at all[4]. Among others OCL can be used to specify *invariants*, *pre-*, and *postconditions*. An invariant is attached to a `Classifier`. The invariant is of type `boolean` and is required to evaluate to `true` for each instance of the `Classifier` at any

[4]There are extensions to the OCL that allow for model modifications (e.g. the imperative OCL package from the QVT standard).

moment of time. The OCL standard does not specify what happens if an invariant evaluates to `false`. A model which violates any invariant simply does not conform to the underlying metamodel. A precondition is an `boolean` expression that is attached to an `Operation`. Preconditions must evaluate to `true` before the attached `Operations` may be executed. Correspondingly, a postcondition is a `boolean` expression attached to an `Operation` that must evaluate to `true` after the execution of the attached `Operation`. Again the standard does not specify what happens when a pre- or postconditions ever evaluates to `false`.

The abstract syntax of OCL (i.e., the metamodel of OCL) is defined on the UML superstructure rather than the UML infrastructure. For supporting MOF OCL provides two packages `BasicOCL` and `EssentialOCL`. As the names imply `BasicOCL` defines a subset of OCL that matches the `Basic` package form the UML infrastructure whereas `EssentialOCL` complements the `EMOF` package. Therefore, `EssentialOCL` is in line with CMOF as well.

3 QVT

As we have motivated in the preceding chapter we aim at founding our own approach on actually existing OMG standards. The upcoming Query / View / Transformation (QVT) standard is OMG's approach for model integration. QVT complements the already introduced metamodeling standards MOF and OCL. In order to increase the usability and acceptability of our own approach we aim at integrating our approach with the QVT standard where reasonable by means of syntax and concepts.

We start by explaining the running example we want to use throughout this work. After that we introduce OMG's initial Request of Proposals (RfP) which asked for proposals for a model integration approach. Relying on the running example we then present the resulting QVT standard in detail. Finally, we discuss shortcomings of the current QVT standard which we want to address and avoid in our own approach.

3.1 Running example

In 2005 the *Model Transformation in Practice Workshop*[1] (MTiP) was collocated with the *MoDELS* conference[2]. This workshop has been organized by members of the QVT related community as well as by members of the (triple) graph grammar community. The corresponding *Call for Papers* (CfP)[3] asked the participants to tackle the following task using their favorite model integration solution. The aim was to compare the used approaches with each other. The same task is addressed by the upcoming QVT standard as the running example. Therefore, this task can be regarded as an official benchmark for QVT related approaches.

The task deals with the integration of a class diagram with a database schema. In fact the task only requires the transformation of a given class diagram into

[1] http://sosym.dcs.kcl.ac.uk/events/mtip05/
[2] http://www.cs.colostate.edu/models05/
[3] http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf

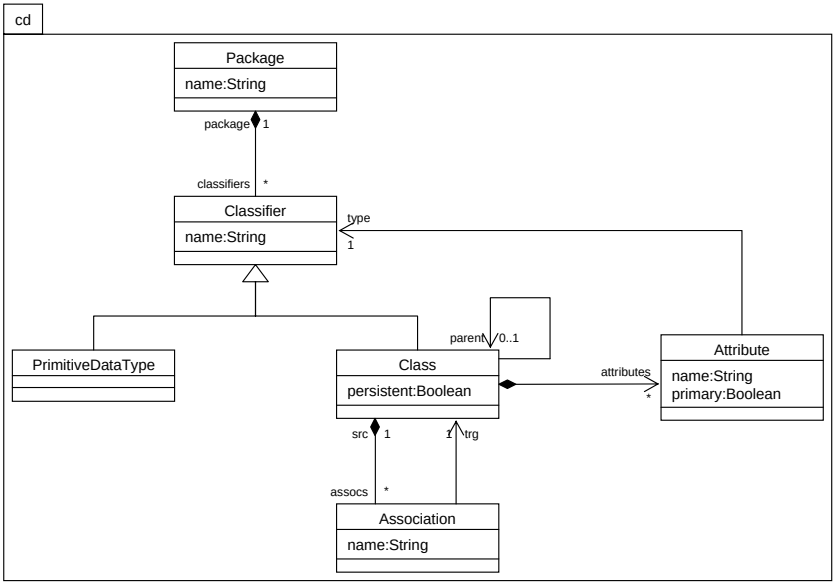

Figure 3.1: Metamodel for class diagrams

a corresponding database schema and not vice versa. Nevertheless, we aim at addressing all integration scenarios as motivated in Chapter 1. The CfP provides MOF metamodels for simple class diagrams (c.f. Figure 3.1) and database schemas (c.f. Figure 3.2). For clarification purposes we have done some minor changes to the provided metamodels.

Basically, a `Package` of a class diagram has a `name` and contains `Classifiers`. A `Classifier`[4] has a `name` and can be a `PrimitiveDataType` or a `Class`. A `Class` owns an arbitrary number of `Attributes`. `Classes` can be marked as `persistent` in order to express whether the instances of a `Class` are made persistent by the corresponding database. Furthermore, each `Class` may be related to another `Class` which represents the `parent` by means of a generalization relationship. Finally, `Classes` may

[4] In UML `Classifier` would be marked as `abstract`. However, in MOF this concept does not exist.

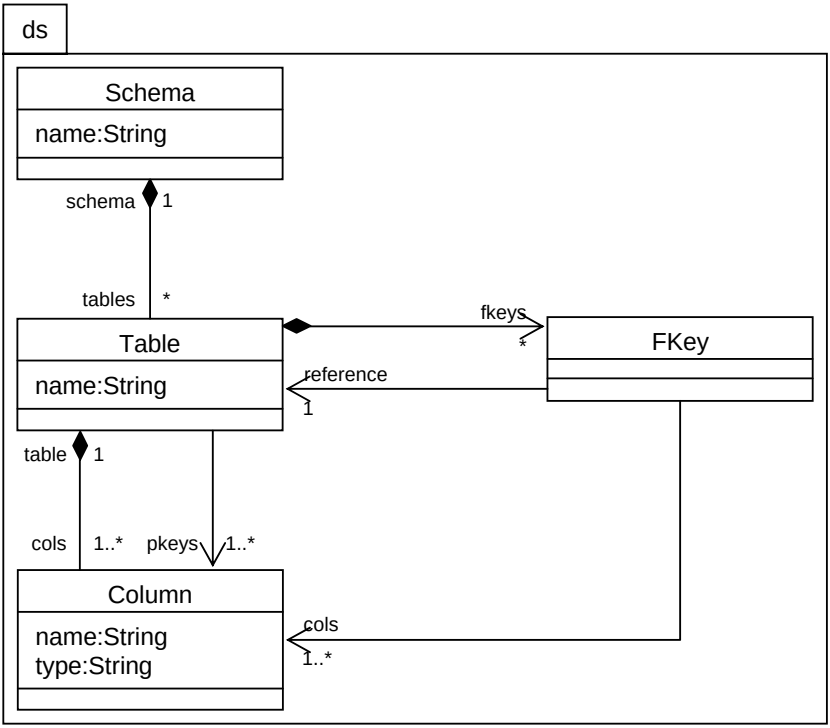

Figure 3.2: Metamodel for database schemas

be related to each other by `Associations`. Each `Attribute` has a `name`. `Attributes` can be marked as `primary` in order to express that the regarded `Attribute` should be used as a primary key in the corresponding database. Furthermore, an `Attribute` has a `type` which is a `Classifier` (i.e., a `PrimitiveDataType` or a `Class`). An `Association` is related to a source (`scr`) and a target (`trg`) `Class`. Each `Association` has a `name`. A separate OCL constraint provided by the CfP demands that each `Class` owns at least one `Attribute`. Additionally, at least one of the owned `Attributes` must be marked as `primary`.

(Database) Schemas (c.f. Figure 3.2) have a name and consist of Tables. Tables have names and consist of Columns. Furthermore, a Table may designate a subset of its Columns as primary keys (pkeys). Finally, a Table has an arbitrary number of foreign keys (fkeys). Columns simply have a name and a type. Each FKey refers to another Table and designates a subset of the Columns of the Table that owns the FKey as foreign keys.

Besides the metamodels of the to be integrated models the CfP provides a number of rules how to realize the intended integration:

R1. Persistent classes in a given class diagram should be transformed into a corresponding table in the database schema. The name of the table should match the name of the class.

R2. Persistent classes that inherit from another class should correspond to the same table in the database schema to which the parent class corresponds to.

R3. Non-persistent classes should not be transformed at top-level.

R4. Attributes that have a primitive data type should be transformed into a column. The name of the column should match the name of the attribute. The type of the column should match the type of the attribute.

R5. An attribute a whose type is a persistent class c should be transformed as follows. For each primary key attribute of c the table that corresponds to the class that owns a should have a column. The column should be named *name_transformed attr*. Thereby, *name* denotes the name of c. The set of created columns should be marked as constituting a foreign key. The foreign key should refer to the table that corresponds to the persistent class c.

R6. An attribute a whose type is a non-persistent class c should be transformed as follows. For each attribute of c whose type is a primitive type the table that corresponds to the class that owns a should have a column. The column should be named *name_transformed attr*. The type of the column should match the type of a. For each attribute of c whose type is either a persistent or a non-persistent class the preceding rules should apply recursively.

Because of rule R2 the transformation from a given class diagram into a database schema and back into a new class diagram possibly looses information. The transformation cannot recreate an inheritance hierarchy for classes created from a single table without additional information or user interaction. Rather, for each table

in the database schema only one class will be created in the class diagram which owns all attributes created from the columns of the table.

In order to clarify the rules and show a simple test case the CfP provides a to be transformed class diagram (c.f. Figure 3.1 and the corresponding database schema (c.f. Figure 3.2). The provided test case is too simple to test all rules. On the one hand the test case does not include an inheritance hierarchy. On the other hand no class contains an attribute whose type is a non-persistent class which in turn has an attribute whose type again is a non-persistent class. In order to test our own approach we will, therefore, transform the simple test case first and then come up with a more sophisticated transformation example in order to demonstrate the application of the more complex rules.

3.2 Request For Proposal

In 2002 the OMG published a Request for Proposal (RFP) [OMG02] which "addresses a technology neutral part of MOF and pertains to: 1. Queries on models. 2. Views on metamodels. 3. Transformations of models." Whilst the first chapters of the RFP only provide general information on the OMG as well as on the process of evaluating and adopting proposals, chapter 5 and 6 describe general and specific requirements which must be fulfilled by submissions to this RFP.

The general requirements are:

G1. Models used in a submission should be expressed using OMG's modeling languages (e.g. UML, MOF).

G2. Any model written in such a language should be accompanied by a matching XMI representation.

G3. If a submission utilizes both PIMs and PSMs the submission should provide mappings between the PIMs and the corresponding PSMs.

G4. A submission should provide all relevant assumptions and context information.

G5. Each submission must clarify which features are mandatory and which are optionally for implementation purposes.

G6. Submissions are encouraged to reuse existing (OMG) standards rather than introducing entirely new models specifying already existing functionality.

G7. A submission should justify any modifications it requires to existing OMG specifications. Moreover, a submission should aim at upward compatibility with existing standards.

G8. Submissions should address reusability issues by factoring out functionality that could be used in various contexts.

G9. Although submissions should reuse functionality from already existing specifications the number of dependencies should be as small as possible.

G10. A submission should not constrain implementations more than necessary.

G11. Submissions should be compatible with ISO's Reference Model of Open Distributed Processing.

G12. Each submission should discuss whether it can be used in environments that require security issues.

G13. Submissions should specify to which degree they provide internationalization support.

The mandatory specific requirements are:

M1. Each submission should provide a language for querying models in order to filter model elements and to select model elements as source for transformations.

M2. Submissions should provide a language for specifying transformations that transform a source model conforming to one metamodel into a target model conforming to another (or the same) metamodel.

M3. Each submission should define the abstract syntax of its query, view, and transformation languages as MOF 2.0 metamodels.

M4. The proposed transformation language of each submission should support the automatic transformation of a source into a target model.

M5. Furthermore, the regarded transformation language should provide means to create views of metamodels.

M6. Submissions should support the incremental propagation of changes from a considered source into the corresponding target model.

M7. Each submission should assume that the metamodels of the regarded models are defined using MOF 2.0.

Finally, the optional specific requirements are:

O1. Submissions may support transformation specifications that can be executed bidirectionally (i.e., source-to-target as well as target-to-source transformations).

O2. Submissions may calculate and utilize traceability information between elements of the source and elements of the target model.

O3. Submissions may provide means for reusing and extending generic transformation specifications.

O4. Submissions may provide transactional mechanisms (i.e., commit and rollback) for (part of) transformations.

O5. Submissions may support the consideration of external data that resides neither in the source nor in the target model.

O6. Submissions may support the transformations in the case that source and target model coincide (i.e., in-place transformations).

We aim at designing our own approach in a such a way that the result could have been a promising submission to OMG's RFP. Actually, when we started designing our approach the deadline for submissions to the QVT-RFP was already due. Nevertheless, we want to regard as many requirements of the RFP as possible. As the current QVT standard which we will introduce later in this chapter suffers from a number of shortcomings our approach can be seen as a proposal of how to deal with some of these shortcomings.

Particularly, our approach considers the requirements as follows. Regarding Requirement G1 it is unclear what is meant with *model*. In accordance with Requirements M3 and M7 we present the metamodel of our model integration approach as a MOF 2.0 metamodel and assume that the to be integrated models themselves conform to MOF 2.0 metamodels. Concerning Requirement G2 we rely on the Java Metadata Interface (JMI) standard [Sun02] for implementation purposes. This standard provides XMI readers and writers for a given MOF metamodel. Thereby, XMI is XML-based textual representation destined for the serialization of models. Requirements G3 does not apply to our approach. Naturally, we want to adhere to Requirement G4 as far as possible. Nevertheless, this work is not a complete technical reference. As we do not distinguish between mandatory and optional features of our approach Requirement G5 does not apply to our approach. Regarding Requirement G6 we state that we plan

to adopt a number of convenient and user-friendly features of the current QVT standard. However, we intentionally do not found our approach directly on the QVT standard. On the one hand we want to get rid of the shortcomings of the QVT standard. On the other hand our approach relies on a formal foundation that does not reside in OMG's world of metamodeling, yet. As we do not modify any existing OMG specifications Requirement G7 does not apply to our approach. Furthermore, we do not aim at regarding Requirement G8. Requirements G9 and G10 are too generic for proving adherence to them. Requirements G11, G12, and G13 are just out of scope for our approach.

Regarding Requirements M1 and M2 we state that the left-hand sides of our TGG rules which we will introduce later on in detail constitute the query part while the right-hand sides constitute the transformation part of our model integration language. As (semi-)automatic model integration support is one of the key goals of our approach we strongly want to adhere to Requirement M4. Concerning Requirement M5 we admit that our approach does not aim at dedicated view creation support. Nevertheless, in Section 11.1 we comment on this issue. Surprisingly, the current QVT standard intentionally disregards views as well. In accordance to Requirement M6 our approach aims at the incremental propagation of model changes. As we will point out in Chapter 8 our current implementation of our approach does not yet support incremental updates.

Since we aim at a declarative model integration approach we want to adhere to Requirement O1. As the maintenance of traceability information is another key goal of our approach we also want to adhere to Requirement O2. Furthermore, we want to incorporate means for reuse and extensibility in accordance to Requirement O3. We do not consider Requirement O4 in this work. We do not want to allow for any external data as proposed in Requirement O5. Finally, we do not support in-place transformations as proposed in Requirement O6. For in-place transformations we rely on graph transformations as introduced in Chapter 4 as a more appropriate approach for this purpose.

3.3 Specification

There have been a number of submissions to OMG's QVT-RFP from various groups with partners from companies (e.g. IBM, Sun) as well as from universities (e.g. Kings College London, University of Paris VI, University of New

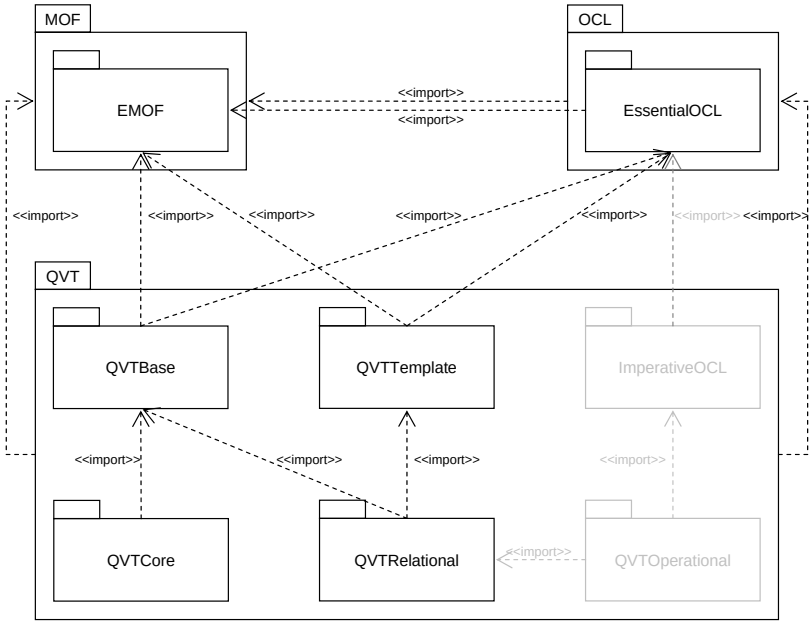

Figure 3.3: Package structure of QVT

York). Meantime, the OMG has adopted a specification which currently is under finalization.

The specification states that it depends on OMG's MOF 2.0 and OCL 2.0 standards. Figure 3.3 illustrates these dependencies. The specification defines three (sub-)languages called *Relational*, *Core*, and *Operational*. The Relational and the Core languages allow for the declarative specification of queries and transformations, whereas the Operational language provides an imperative approach. *Declarative* means that the specification describes desirable situations (i.e., both regarded models are consistent with each other) but does not state how to reach these situations. In contrast an *imperative* specification explicitly provides operations that can be invoked in order to keep two regarded models consistent with each other.

The Relational language aims at user-friendliness and supports complex object pattern matchings. Traceability links between model elements are maintained implicitly. The Relational language provides a graphical and a textual concrete syntax. In contrast, the Core language is defined using minimal extensions to EMOF and OCL. Traceability links are explicitly specified, maintained, and dealt with as any other model element. There is only a textual concrete syntax available for the Core language. Since the Core language is quite simple its semantics can be defined quite easily. The semantics of the Relational language is given by a transformation that transforms a given Relational specification into a semantically equivalent Core specification. The Relational and the Core languages are equally powerful. As already mentioned above the Operational language realizes an imperative approach. Therefore, the Operational language rather matches operational graph grammar-based approaches than declarative triple graph grammar-based approaches. Thus, we disregard the Operational language for the remainder of this work. Nevertheless, we should mention that QVT provides an extension to OCL called *imperative OCL* which is used by the Operational language. In contrast to OCL which only supports side-effect free model queries imperative OCL provides functionality to intentionally modify models.

3.3.1 Basic concepts

The *QVTBase* package (c.f. Figure 3.4) contains the basic concepts of QVT which are used throughout the definition of the three sublanguages. The central construct of QVT is called *Transformation*. A transformation describes how to transform a set of typed models into another. To this end a transformation contains a number of rules. At runtime a transformation is executed in a certain direction which specifies which models are considered as source and which model is considered as target of the transformation. A *typed model* as an input of a transformation is a model that conforms to a metamodel written in EMOF. A *rule* specifies how model elements of typed models are related with each other. To this end each rule contains a number of domains. A *domain* specifies which elements of a typed model are regarded by the corresponding rule. A domain can be marked as checkable or enforceable. For domains that are marked as *checkable* the containing rule must check whether the model elements specified by the domains exist and report missing elements. In contrast a rule must ensure the existence of all model elements of domains that are marked as *enforceable* by modifying the corresponding typed model appropriately. Furthermore, the *QVTBase* pack-

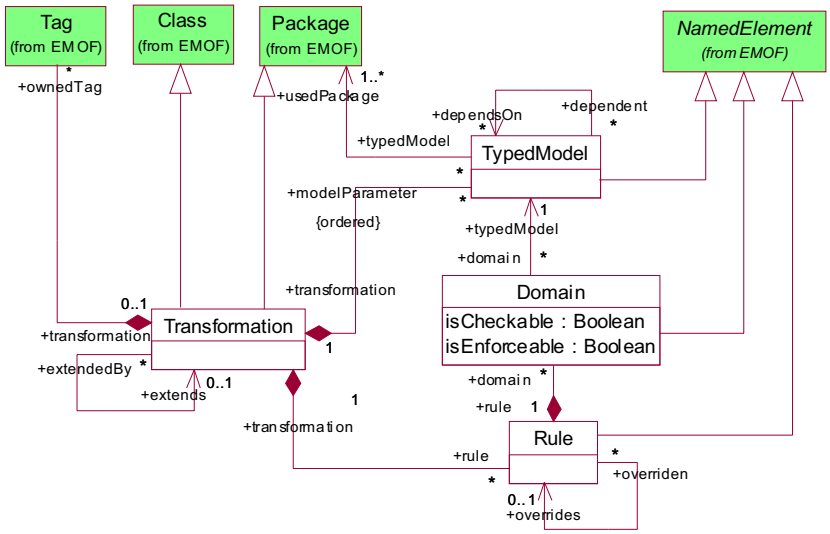

Figure 3.4: QVTBase package taken from [OMG05b]

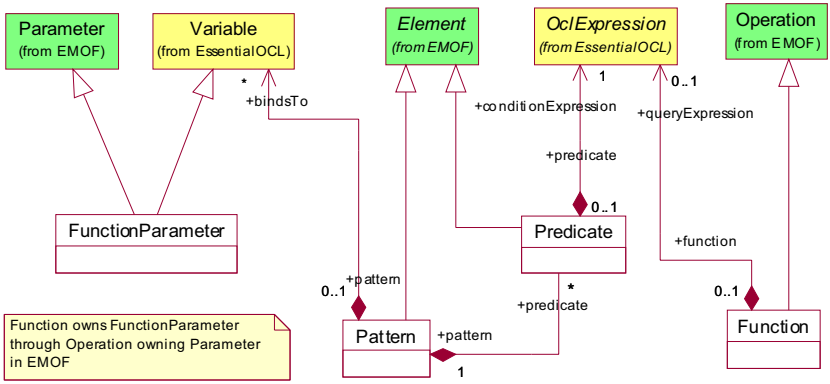

Figure 3.5: QVTBase package taken from [OMG05b] *(cont.)*

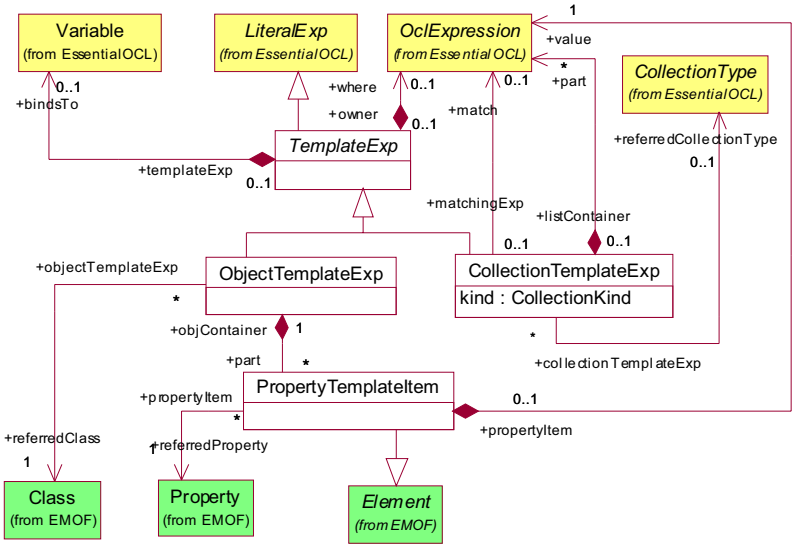

Figure 3.6: QVTTemplate package taken from [OMG05b]

age (c.f. Figure 3.5) introduces the concept of patterns. A *Pattern* declares a number of variables which are to be bound by matching the pattern in a typed model. Patterns may have *predicates* that are boolean expressions that constrain the matching of a pattern in a typed model. Besides patterns transformations own functions. A *function* is an operation which is free from any side-effects (i.e., a function may not perform any modifications). Therefore, functions represent queries. Each function may be provided with a number of *function parameters*.

The *QVTTemplate* package (c.f. Figure 3.6) basically introduces the concept of template expressions. A template expression matches one (*object template expression*) or multiple (*collection template expression*) elements in a typed model. A template expression only matches if the attached *where* expression holds as well. A template expression may have a number of property template items. A *property template item* specifies constraints on the values of attributes of the regarded model elements.

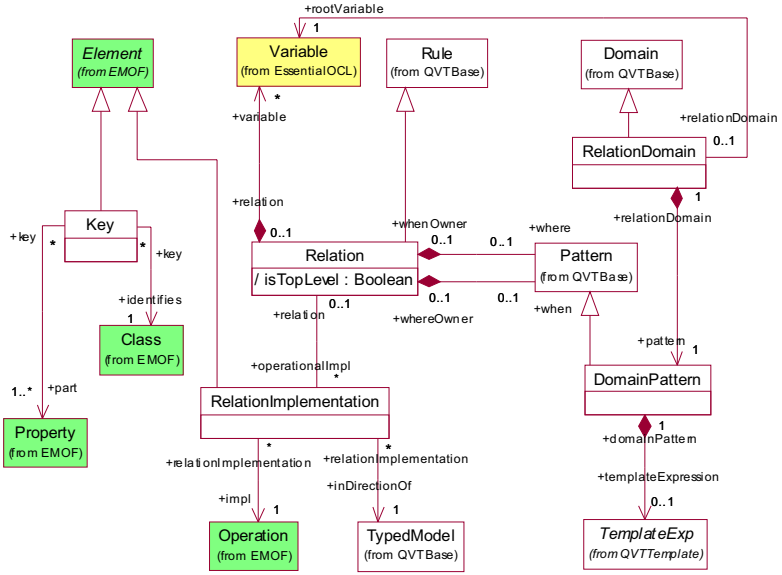

Figure 3.7: QVTRelation package taken from [OMG05b]

3.3.2 The Relational language

The *QVTRelation* package (c.f. Figure 3.7 incorporates the concepts from the *QVTBase* and the *QVTTemplate* packages and provides specializations for them. The basic concept of the Relational language is a relation. A *relation* is a specialization of a rule. Each relation declaratively specifies which model elements relate to which model elements. To this end a relation declares at least two relation domains, a when- and a where-pattern. A *relation domain* is a specialization of a domain. Each relation domain is provided with a *domain pattern* which is a specialization of a pattern and is to be matched in the corresponding typed model. Furthermore, each relation domain has a distinguished variable that is called *root variable*. The *when*-pattern of a relation acts like a precondition for the relation. That means that the relation must only hold for situations when the *when*-pattern holds as well. In contrast the *where*-pattern is required to hold when the relation holds. That means that a relation can invoke other relations using where-

patterns. A relation that is not invoked by any other relation is called a *top-level relation*. Since all non-top-level relations are directly or indirectly invoked by top-level relations all models are consistent when all top-level relations hold. Besides the declarative specification each relation may be provided with a relation implementation. A *relation implementation* is a black-box implementation which operationally enforces the corresponding relation when the relation does not hold. Finally, the *QVTRelation* package introduces the concept of keys. A *key* is a set of attributes of a class that uniquely identify instances of that class. Keys are used when a transformation is executed in enforcement mode. Basically, there are two possibilities how to enforce consistency when the target model violates relations. On the one hand the transformation can delete all inconsistent model elements and create consistent ones instead. On the other hand the transformation can modify the inconsistent model elements appropriately. In order to determine which repair action should be taken keys are used in order to identify model elements that can be modified instead of being deleted and recreated.

Regarding our running example of integrating a class diagram with a corresponding database schema the declaration of a transformation in QVT Relational looks as follows:

```
transformation cdds_integration(cd:cd_metamodel,
                                db:db_metamodel} {
```

The declared transformation is called `cdds_integration`. The tranformation can be invoked on two models, one of which conforms to the `cd_metamodel` metamodel while the other conforms to the `cd_metamodel` metamodel.

For instance the declaration of a relation looks as follows:

```
relation PackageToSchema {
    domain cd p:Package { name = pn }
    domain db s:Schema  { name = pn }
}
```

The name of this relation is `PackageToSchema`. The relation declares two domains `p` of type `Package` from the `cd_metamodel` metamodel and `s` of type `Schema` from the `db_metamodel` metamodel. Additionally, there is a third variable `pn`. This variable implicitly expresses that the `name` of `p` should

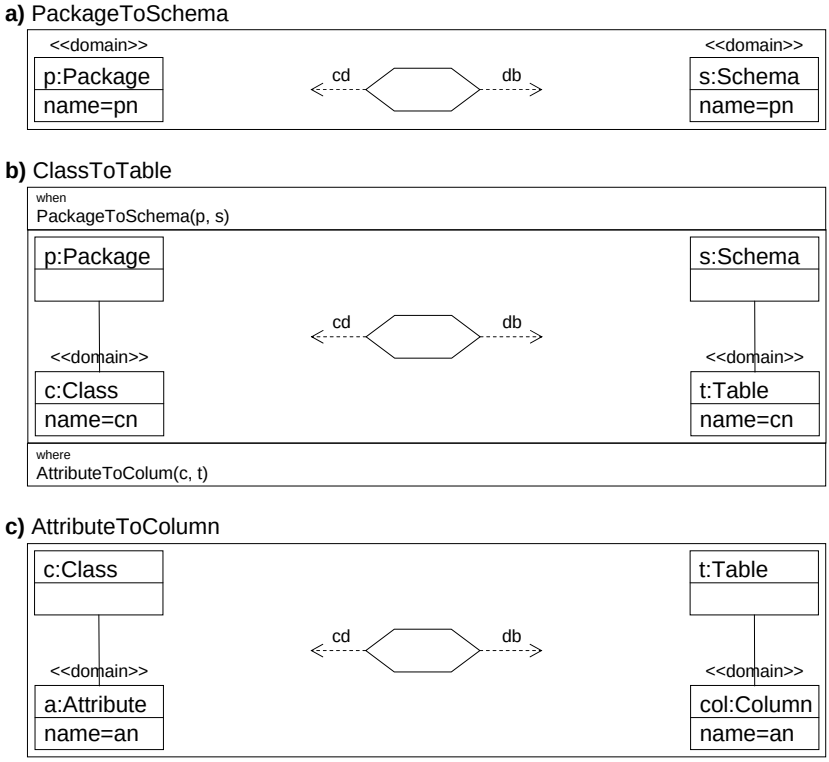

Figure 3.8: Examples of QVT's graphical syntax

match the `name` of `s`. Figure 3.8a illustrates this relation using the graphical syntax of QVT.

If the transformation `cdds_integration` is invoked in order to check two models for consistency for each `Package p` there must be a `Schema s` such that the `name` of `s` matches the `name` of `p` and vice versa. Otherwise, both models are not consistent with each other.

If the transformation `cdds_integration` is invoked in order to enforce consistency of two models one model must be designated as the source and the

other as the target model beforehand. Let us assume that cd refers to the source and db refers to the target model. If there exists a Package p in cd such that there is no Schema s with a matching name in db the Transformation will modify db in order to enforce consistency. To this end the transformation either can change the name of an existing Schema or the Transformation can create a new Schema with the matching name. In order to select one of these possibilities the Relation language provides the key construct which will be described later on. Finally, if there is a Schema s which name does not match the name of any Package p the transformation deletes s.

Let us now look at the declaration of a second relation ClassToTable:

```
relation ClassToTable {
    domain cd c:Class { namespace = p,
                        name = cn }
    domain db t:Table { schema = s,
                        name = cn }

    when {
        PackageToSchema(p, s);
    }

    where {
        AttributeToColumn(c, t);
    }
}
```

The graphical syntax of this relation is shown in Figure 3.8b. Similar to relation PackageToSchema this relation demands that for each Class c in cd there should be a Table t in db such that the name of t matches the name of s and vice versa. Additionally, the when-clause states that the name of t must only match the name of c if the namespace p of c relates to the schema s of t. Finally, the where-clause states that if c relates to t then the Attributes of c must relate to the Columns of t according to the following relation:

```
relation AttributeToColumn {
    domain cd a:Attribute { class = c,
                            name = an }
```

```
    domain db col:Column  { table = t,
                            name = an }
}
```

Figure 3.8 depicts this relation in QVT's graphical syntax.

Finally, the declaration of keys looks as follows:

```
transformation cdds_integration(cd:cd_metamodel,
                                db:db_metamodel} {
    key Table(name, schema);
    key Column(name, owner);
    ...
}
```

That means that the relation `ClassToTable` will create a new `Table` when the transformation is executed in `enforce` mode and a `Table` with matching `name` and `schema` does not already exist. Accordingly, a new `Column` will only be added if there is no `Column` with matching `name` and `owner`.

Expressions used in relations must adhere to the following restrictions in order to guarantee the executability of a transformation. Arbitrary expressions would require a sophisticated constraint solver which is hard to implement. Therefore, it must be possible to rearrange expressions in the `when`-clause, the `where`-clause, and the source domains such that:

1. *`object.property = variable`*
   where `variable` is an unbound variable and `object` is a variable that already has been bound to an object beforehand.
2. *`object.property = expression`*
   where `object` has already been bound to an object beforehand and `expression` does not contain any unbound variables.
3. There are no further expressions that contain unbound variables.

Furthermore, for the target domain holds that:

4. *`object.property = expression`*
   where `object` has already been bound to an object beforehand and `expression` does not contain any unbound variables.
5. There are no further expressions that contain unbound variables.

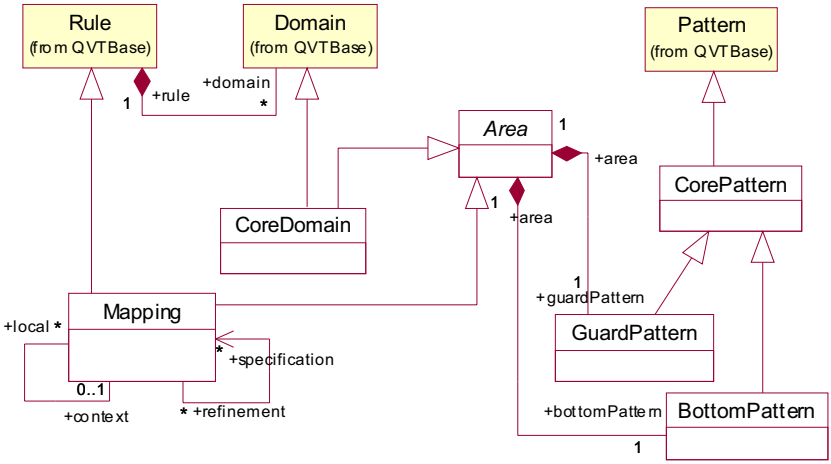

Figure 3.9: QVTCore package taken from [OMG05b]

3.3.3 The Core language

As already mentioned above the Core language in contrast to the Relational language which aims at user-friendliness is simpler and more technical. The basic concept of the Core language is called mapping (c.f. Figure 3.9). A *mapping* is a specialization of a QVT rule. Each mapping consists of a middle area and an arbitrary number of core domains. An *area* consists of a guard and a bottom pattern. A *guard pattern* is a pattern which is to be matched in a typed model. Thereby, the matching of a guard pattern provides a binding for the variables of the pattern without modifying the typed model. A *bottom pattern* is only matched when its corresponding guard pattern has been successfully matched. The bottom pattern may use variables which already have been bound by the matching of the guard pattern. In contrast to the guard pattern a bottom pattern may modify the underlying typed model. A *core domain* is an area that is assigned to a certain typed model. Since core domains inherit from domains they can be marked as checkable or enforceable as explained above. As Figure 3.10 shows bottom patterns consist of an arbitrary number of realized variables, assignments, and enforcement operations. *Realized variables* can be used in order to create or delete model elements. *Assignments* are used to modify attribute values of matched model elements. An

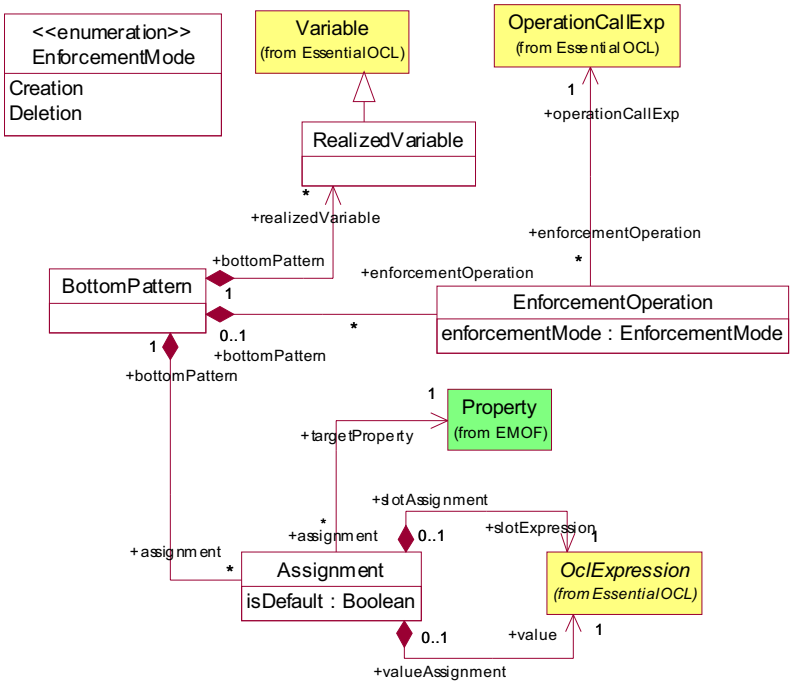

Figure 3.10: QVTCore package taken from [OMG05b] *(cont.)*

enforcement operation is invoked in a certain *enforcement mode* (i.e., creation or deletion mode). In creation mode the enforcement operation is supposed to create new model elements in order to recover model consistency. Whereas, in deletion model the enforcement operation is supposed to delete model elements.

Figure 3.11 clarifies the decomposition of a mapping into areas and patterns. *Domain L* contains the patterns to be matched in one model. *Domain R* contains the patterns to be matched in a second model. *Middle Area* contains the patterns to be matched in the mapping model. To recall in the Core language correspondence links are explicitly dealt with in an own model. The *guard patterns* have to be matched in their corresponding model in order to allow for the matching of

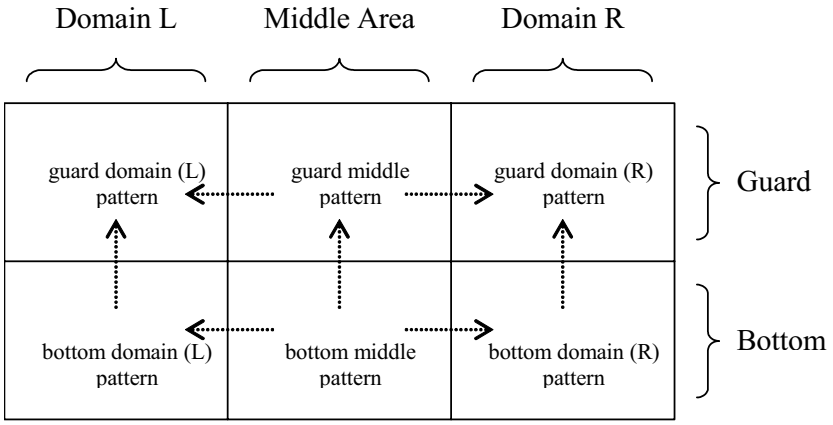

Figure 3.11: Patterns in a QVT core mapping taken from [OMG05b]

the attached *bottom* patterns. The *bottom patterns* may modify their underlying models and, thus, realize the desired model integration.

3.4 Shortcomings

The current QVT standard suffers from a number of shortcomings. The presented metamodels are too general and do not reflect the intended semantics given in the textual description. For instance in Figure 3.4 a transformation inherits from `Class` and `Package` from EMOF. The rationale behind this is that a transformation as a package defines a namespace for its contained elements. Furthermore, a transformation as a class can have attributes. However, as a package a transformation may contain arbitrary packageable elements as plain classes for instance. On the one hand it is questionable whether this is really intended and on the other hand the standard does not provide semantics for plain classes as part of a transformation. As a class a transformation may have operations. Again it is questionable whether this is intended and what that means. Therefore, the standard should provide constraints in order to restrict the metamodels. These constraints could be given as plain text, as OCL constraints, or by using CMOF instead of EMOF

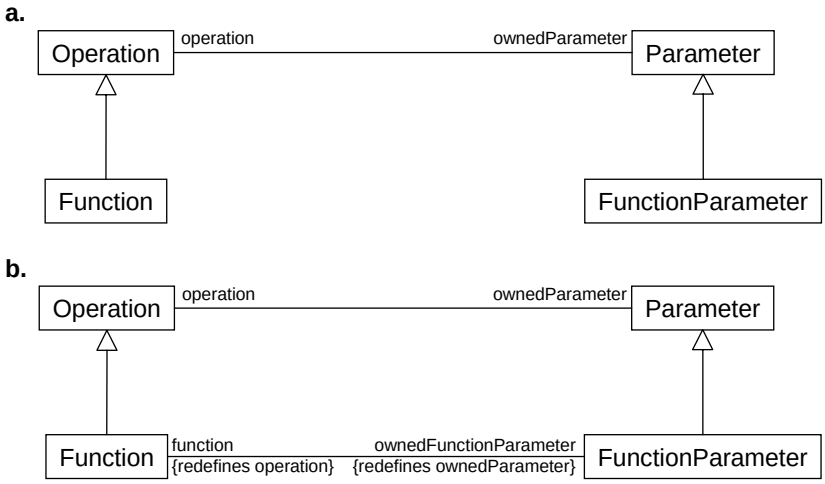

Figure 3.12: Part of QVT's metamodel **a.** incomplete, **b.** completed

which provides concepts as *subset* or *redefines*. Figure 3.12 illustrates this. Figure 3.12a. represents a situation from Figure 3.5. The standard says that "Function owns FunctionParameter through Operation owning Parameter in EMOF". However, the preented metamodel does not reflect this. Since `Function` inherits from `Operation` a function owns parameter. Since `FunctionParameter` inherits from `Parameter` a function can have function parameter besides any other parameters. Furthermore, since an operation owns parameters function parameters may be attached to arbitrary operations. This is not what is intended. Using the concept of *redefines* from CMOF the fixed metamodel looks as depicted in Figure 3.12b. The redefinitions ensure that a function can only have parameters that are function parameters. The other way round a function parameter can only be attached to a function. The drawback of this solution is that it requires the QVT standard to use the more complex CMOF rather than EMOF for its definition. Another solution is to use the following OCL constraints:

```
context Function
inv: ownedParameter->forAll( p |
         p.oclIsKindOf( FunctionParameter ))
```

```
context FunctionParameter
inv: operation.oclIsKindOf( Function )
```

The first constraint concerns class `Function`. The constraint states that all `ownedParameters` of `Function` in fact are `FunctionParameters`. Correspondingly, the second constraint concerning `FunctionParameters` ensures that the `Operation` a `FunctionParameter` is attached to in fact is a `Function`. The advantage of these constraints is that they do not require QVT to switch to CMOF. The disadvantage is that textual constraints are less elegant than using *redefines* from CMOF. Furthermore, the usage of *redefines* improves the readability of the metamodels. For instance Figure 3.12b explicitly depicts the relationship of `Functions` and `FunctionParameters`. In contrast in Figure 3.5 this relationship is only implicitly given and not depicted. This exacerbates the understandability of QVT's metamodel.

Another issue is the application of the *checkonly* and *enforce* concept. The QVT standard states that domains can be marked as *checkonly* or *enforce*. However, the standard does not require a domain to be marked. Furthermore, the standard does not describe in which way a domain that is not marked is dealt with (e.g. by defining a default value). On the one hand it would be reasonable to regard *checkonly* as default. A transformation may not modify models by matching domains marked as *checkonly* and, therefore, cannot accidentally corrupt the regarded models. On the other hand it would be reasonable to regard *enforce* as default since usually transformations are intended to modify models. Regardless which alternative is preferable the QVT standard should specify a default. Otherwise two standard-compliant implementations might treat a given transformation differently. Another solution is to make the marking of domains as *checkonly* or *enforce* mandatory.

Furthermore, it is questionable whether it is necessary to specify *checkonly* and *enforce* at the fine-grained level of rules rather than on the level of transformations. Certainly, the specification at rule level is more flexible and more expressive. Nevertheless, it exacerbates the understandability of a transformation as a whole since some rules might modify the underlying models whilst other rules only perform checks in enforcement mode. Altogether, practice must prove whether the specification of *checkonly* and *enforce* is really required at a fine-grained level.

The key concept introduced by the QVT standard suffers from the following fact. The definition of keys by the specificator of a transformation requires them to know which sets of attributes of a certain class in a given metamodel constitute keys. Since the specificator of a transformation might differ from the specificator of the considered metamodel the definition of keys in fact should be included in the metamodel. Unfortunately, MOF does not provide support for the definition of keys. Although (OCL) constraints can be used to ensure that sets of attributes adhere to the requirements for keys (i.e., the values of attributes that constitute a key uniquely identify an instance of the corresponding class) this information cannot be externally accessed by the specificator of a transformation. Therefore, keys defined by the specificator of a transformation might violate the requirements of keys resulting in erroneous transformations. Thus, the concept of keys introduced by the QVT standard is merely an unsatisfactory crutch. Since keys are useful not only for model transformation purposes MOF need to be extended by an adequate key concept.

Finally, in our opinion the restrictions the QVT standard puts on expressions used in rule patterns are too strong at conceptual level. Admittedly, less restrictions might make it difficult to implement the standard and requires the use of a powerful constraint solving mechanism in general. Nevertheless, implementational issues should not affect the standard at conceptual level. Furthermore, the QVT standard says that "It should be possible to organize the expressions that occur in the *target domain*, into a sequential order that contains only the following kinds of expressions: ...". This statement implies a certain direction of the desired model transformation. However, in the context of declarative QVT specifications the direction of transformation will be chosen at runtime not at specification time. Therefore, the statement undesirably requires the specificator to virtually interpret their declarative specification in mind at specification time.

Being aware of the identified shortcomings of QVT we aim at appropriately addressing them in our approach. First of all, we specify the metamodel of our approach by utilizing the CMOF standard rather than EMOF. By doing so we can rely on the concepts of *subsets* and *redefines* for conveniently specifying a more correct and complete metamodel.

Furthermore, we initially do not adopt the concepts of *checkonly* and *enforce* from the QVT standard. As we will see later we derive operational rules from declarative rules, which will never modify the source and the target model at the same time. Rather, model transformation will be executed in a certain user-controlled direction. In our approach *checkonly* and *enforce* could be used in order

to restrict the set of to be derived operational rules. As the rule derivation will be performed automatically it is questionable whether restricting the set of derived rule is really useful in practice. Definitely, we are convinced that *checkonly* and *enforce* should not be used at the fine-grained level of rules.

Regarding the key concept as proposed by the QVT standard, we think that such a key concept should be defined by the MOF standard rather than by any model integration approach. As we will see later our approach distinguishes elements that are to be created from elements that already exist. Furthermore, our approach provides detailed context information that enables us to uniquely identify an element. Thus, we do not really need a key concept. Therefore, we consider the definition of an own key concept as out of scope for our work.

Finally, at conceptual level we will not put any restrictions on expressions used in rule patterns. However, our implementation will not incorporate the needed sophisticated constraint solving mechanism and, thus, will only be able to deal with a restricted subset of possible expressions.

4 Graph Grammars

In this chapter we present the basics of graph grammars as far as they concern our approach. On the one hand graph grammars constitute the formal foundation of triple graph grammars in general and our approach in particular. On the other hand from each declarative triple graph grammar rule a number of operational graph grammar rules is derived (cf. Chapter 7) which can be applied in order to realize the desired model integration. Furthermore, for code generation purposes we rely on a meta-case tool that is able to generate Java code from graph grammar rules (cf. Chapter 8). Therefore, graph grammars can be regarded as intermediate code between the declarative triple graph grammar specification and executable (Java) code.

We start by briefly giving an introduction to graph grammars. After that we explain basic and more sophisticated elements of graph grammars and informally explain their semantics. Finally, this chapter concludes by introducing the idea of pair and triple graph grammars on which our work relies on.

4.1 String grammars

Graph grammars have been introduced in the late sixties by Pfaltz, Rosenfeld [PR69] as an extension to string grammars. String grammars are a well-known concept and are widely used for the specification of textual (programming) languages (e.g. the Java language). A language is a (usually indefinite) set of sentences (e.g. syntactically correct Java programs). Each sentence is made from a given set of words (e.g. reserved Java words, identifiers, etc.). A grammar consists of a set of rules (e.g. written in BNF) that describe in which way words can be combined in order to make sentences that belong to the regarded language.

The following example taken from the Java language specification [GJSB05] specifies the language of decimal numbers that are allowed in Java.

$$
\begin{aligned}
\textit{DecimalNumeral} &::= 0 \mid \textit{NonZeroDigit Digits} \\
\textit{Digits} &::= \varepsilon \mid \textit{Digit} \mid \textit{Digits Digit} \\
\textit{Digit} &::= 0 \mid \textit{NonZeroDigit} \\
\textit{NonZeroDigit} &::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

The first rule states that a *DecimalNumeral* either is a `0` or a sequence of *NonZeroDigit Digits*. The second rule specifies that *Digits* are either the empty word, a *digit*, or a sequence of *Digits Digit*. A *Digit* is either a `0` or a *NonZeroDigit*. Finally, a *NonZeroDigit* is either `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`. *DecimalNumeral*, *NonZeroDigits*, *Digits*, and *Digit* are called nonterminal symbols. Nonterminal symbols have to be resolved by the application of further rules. In contrast `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, and `9` are called terminal symbols which cannot be resolved any further. According to this grammar the sentence `6563` is an element of the specified language while `06563` is not.

Basically, string grammars suffer from the following flaws. Firstly, string grammars are restricted to textual languages. Secondly, string grammars can only specify one dimensional languages. The first flaw merely is a matter of taste whether a graphical representation is more comprehensible, more compact, more suitable for a certain task, or anything else than a textual representation. The second flaw is a matter of expressiveness. Both flaws are addressed by graph grammars which in turn come with their own flaws.

4.2 Graph schemas

Similar to a string grammar a graph grammar specifies a language (i.e., a set of graphs). In the literature there are various definitions of the term graph. In the context of this work a graph consists of a set of nodes (or vertices) and a set of edges. Each node and each edge is typed. Each node may carry a number of attributes according to its type. Each edge links two nodes with each other. The type of an edge specifies nodes of which types may be linked by the edge. For a formal definition of graphs the reader is referred to [Sch91]. The types of nodes and edges are specified in a graph schema. A graph must conform to the graph schema in order to be an element of the regarded language. Figure 4.1a. depicts

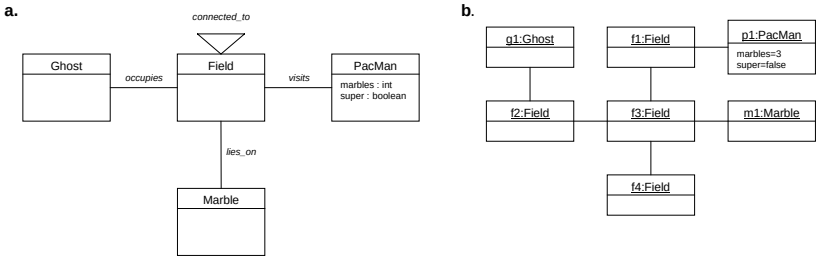

Figure 4.1: Example of **a.** a graph schema and **b.** a conforming graph

an example of a graph schema as presented in [Hec06]. The example deals with a (part of a) Pac Man game. Basically, the game consists of `Fields` that are `connected to` each other in a certain way. A `Field` may be `occupied by Ghosts` and `visited by PacMans`. Furthermore, `Marbles` may `lie on Fields`. Each `PacMan` has an attribute `marbles` which keeps track of the number of `Marbles` that the `PacMan` has collected. Additionally, each `PacMan` has an attribute `super` which states whether or not this `PacMan` is able to kill `Ghosts`. Figure 4.1b. shows a graph that conforms to the graph schema from Figure 4.1a. There are four `Fields f1`, `f2`, `f3`, and `f4` that are `connected to` each other in the depicted manner. `Field f2` is `occupied by` a `Ghost g1`. `Field f1` is `visited by` a `PacMan p1` which has already collected 3 `Marbles` and is not in `super` mode. Finally, a `Marble m1 lies on Field f3`.

Throughout this work it is important to know that the term *schema* from the world of graphs corresponds to the term *schema* from the world of databases as well as to the term *metamodel* from OMG's world of metamodelling. We will uses these terms interchangeably depending on the context.

4.3 Basic rule elements

Besides a graph schema a graph grammar provides a set of graph grammar rules. As string grammar rules do graph grammar rules usually describe the resolution of nonterminal graph elements to graphs that consist of terminal graph elements

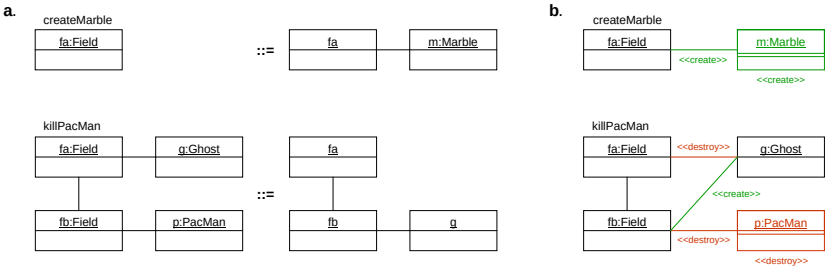

Figure 4.2: Example of **a.** normal graph rules and **b.** collapsed rules

only. Similar to string grammars that have a particular nonterminal symbol called *start symbol* from which the construction starts, graph grammars have a particular rule called *axiom* that will be executed exactly one time at the beginning.

In contrast graph rewriting (transformation) systems provide a set of graph rewriting rules. A graph rewriting system does not consider nonterminal symbols. Rather, graph rewriting rules describe the replacement of (terminal) graph patterns with other (terminal) graph patterns. To be honest we rely on graph rewriting systems rather than on real graph grammars as we are not dealing with nonterminal graph elements.

A graph rewriting rule consists of a left-hand side and a right-hand side. In order to apply the rule on a regarded graph a part of the graph is identified which matches the pattern of the left-hand side of the rule. If such a match cannot be found the rule cannot be applied. If more than one match is found the rule usually is nondeterministically applied to one of the possible matches. Finally, the chosen match is rewritten by the right-hand side of the rule.

Actually, a graph rewriting rule has a third part that denotes graph elements that are kept rather than rewritten. In common approaches (e.g. [Sch91]) this third part is implicitly given rather than explicitly specified. Furthermore, in some approaches (e.g. [Zün01]) left-hand side and right-hand side are written in a collapsed manner in which both sides are concurrently depicted in the same diagram.

Figure 4.2a. gives examples of graph rewriting rules. The first rule demonstrates the matching of an element as well as the creation of new elements. In order to apply rule `createMarble` to the graph of Figure 4.1b. the left hand

side of the rule must be matched. The rule demands to match an element of type Field. There are four possible matches f1, f2, f3, or f4. The nondeterministically chosen match (e.g. Field f4) will be called fa throughout the application of the rule. Since the pattern of the left hand side is completely matched now the pattern will be rewritten by the right hand side. The node fa refers to the just matched Field from the left hand side. Instead of removing and recreating this node it remains untouched. Additionally, a new node m of type Marble will be created and linked to fa by a new edge of type lies_on. Observe that the graph schema does not forbid to add more than one Marble to a regarded Field. We will present means for this issue in the next section.

Rule killPacMan moves a Ghost to a connected Field which is visited by a PacMan which will be removed from the graph. Figure 4.2b. shows the same rules in collapsed style. Elements shown in black denote elements that exist on the left hand side as well as on the right hand side of the rule. Thus, these elements will be matched and kept. Elements shown in green marked with the «create» keyword denote elements that exist only on the right hand side of the rule. Thus, these elements will be created and added to the graph. Finally, elements shown in red marked with the «delete» keyword denote elements that exist on the left hand side of the rule only. Thus, these elements will be matched and then removed from the graph. Since the realization of our approach relies on [Zün01] we stick to the collapsed notation from now on.

4.4 Sophisticated rule elements

Rule createMarble of Figure 4.3 only creates a new Marble and connects it to a Field if there is not already a Marble lying on the considered Field. The pattern object of type Marble shown in this rule is called a *Negative Application Condition (NAC)*.

Rule collectMarble moves a PacMan to a connected Field on which a Marble lies on. The Marble will be removed from the graph. Furthermore, the marbles attribute of the considered PacMan will be increased by one as stated by the provided attribute expression.

In rule killGhosts the depicted attribute condition restricts matches of p to PacMans which super attribute carries the value true. Furthermore, the rule

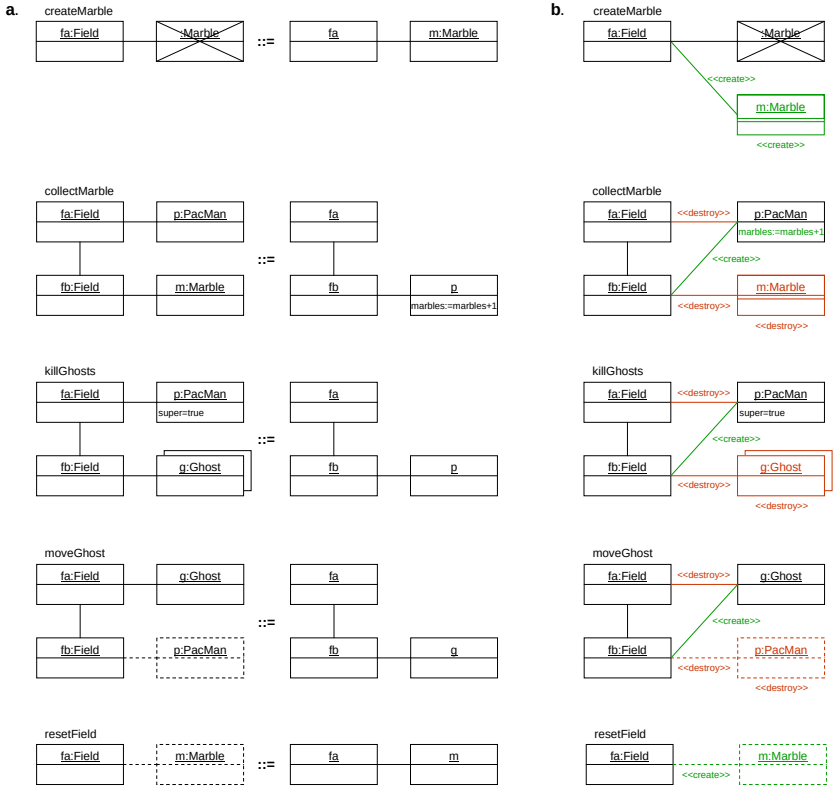

Figure 4.3: Example of **a.** sophisticated graph rules and **b.** collapsed rules

removes the set of `Ghosts` that are `occupying` the `Field` which the matched `PacMan` is about to `visit`.

The next rule `moveGhost` moves a `Ghost` to a `connected Field`. If a `PacMan` is currently `visiting` this `Field` it will be removed from the graph. Nevertheless, this rule matches on `Fields` that are not currently `visited` by a `PacMan` as well.

Finally, the last rule `resetField` illustrates the most sophisticated rule element called *optional create*[1]. The rule matches a `Field`. Furthermore, the rule tries to match a `Marble` that lies on the `Field`. If no such `Marble` can be matched the rule creates a new one and connects it to the `Field`. But if the rule successfully matches a `Marble` lying on the `Field` the rule does nothing. Thus, the application of this rule ensures that finally a `Marble` lies on the matched `Field`. In fact *optional create* can be seen as an abbreviation for a more complex rule that firstly tries to match the optional element and then decides whether or not to create a missing element.

4.5 Pair and Triple Graph Grammars

In 1994 Schürr presented the concept of Triple Graph Grammars (TGGs) [Sch94]. TGGs aim at the declarative specification of model to model integration rules. TGGs are an extension of Pratt's pair grammar approach from 1971 [Pra71]. Pratt's approach implicitly couples two (graph or string) grammars with each other in order to express simultaneous application of grammar rules. The application of such a pair grammar results in two graphs or strings that are said to be consistent to each other.

The following example illustrates Pratt's approach. On the one hand we describe the creation of a very basic database schema which represents a database table. On the other hand we want to create a corresponding SQL statement that results in an equivalent database table.

Figure 4.4a. presents the metamodel of the database schema. The database schema consists of a `Table` that has a `name`. The `Table` owns at least one `Column`. Each `Column` has a `name` and a `type`.

Figure 4.4b. lists information on the intended string grammar. The string grammar uses the non-terminal symbols `S` and `A`. Furthermore, the string grammar uses the terminal symbols `create`, `table`, `(`, `)`, and `,`. Finally, the string grammar is provided with the variables `n` and `t`.

The rules of the graph grammar are depicted in Figure 4.4c., whereas the corresponding string grammar rules are given in Figure 4.4d. The rule `createTable`

[1] To the best of our knowledge this rule element currently is supported neither by PROGRES nor by FUJABA.

creates a `Table` and a first `Column`. It assigns the provided `name` to the `Table`. The rule `addColumn` adds a `Column` to the `Table`. The rule `completeColumn` assigns the provided `name` and `type` to a `Column` that has no `name` and `type`, yet.

According to Pratt the application of a graph grammar rule results in the simultaneous application of the corresponding string grammar rule and vice versa. The simultaneous application of corresponding rules ensures that the resulting models and strings are consistent with each other. Figure 4.5 illustrates the simultaneous application of the rules `createTable("Order")`, `addColumn()`, `completeColumn("order_no", "int")`, and `completeColumn("customer_name", "String")`.

One major drawback of pair grammars is that the correspondence of both graphs or strings is only implicitly given. After derivation of two consistent graphs or strings it is impossible to identify which elements from one graph or string correspond to which elements in the other graph or string. Furthermore, it is not possible to calculate and store any further information about the derivation process. Finally, if one graph or string changes the second graph or string must entirely be rebuilt in order to be consistent to the first graph or string. Therefore, the pair grammar approach is unsuitable for model to model integration purposes when the calculation of additional (traceability) information is wanted or when both models may change independently from each other and changes should be propagated incrementally.

These issues are addressed by TGGs in the following manner. Besides the simultaneous derivation of two models TGGs additionally derive a third model that contains traceability information by means of correspondence links between elements of the first and elements of the second model. Furthermore, these correspondence links may carry additional information calculated during rule application. Finally, the traceability information can be utilized to incrementally propagate changes in one model to the second model.

In the next two chapters we explain TGGs in detail using our running example and discuss our extensions to the original approach from 1994. In Chapter 7 we show how to utilize declarative model integration specifications given as TGGs in order to perform common model integration tasks.

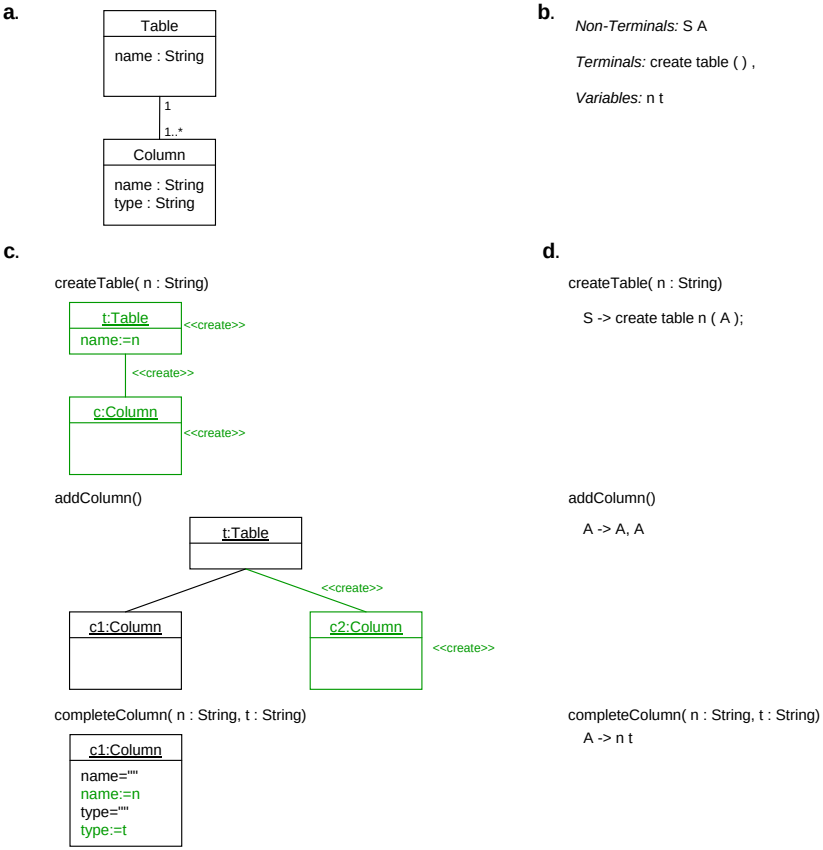

Figure 4.4: Example of a pair grammar

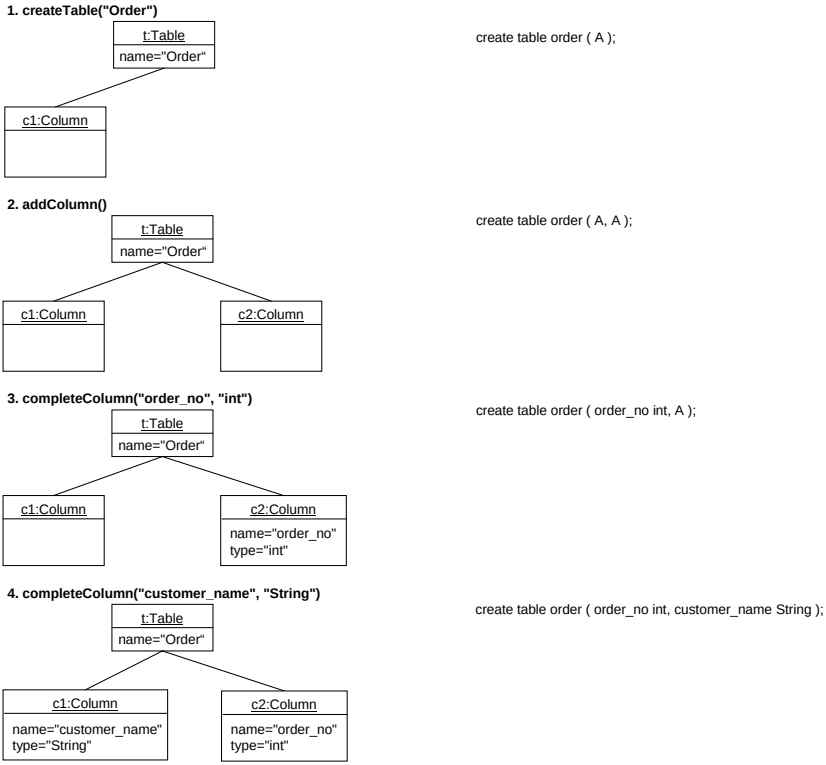

Figure 4.5: Application of a pair grammar

5 TGG schema language

Relying on the initial idea of triple graph grammars as presented in the preceding chapter we now describe our Triple Graph Grammar approach in detail. Furthermore, we discuss how the concepts of our approach can be mapped to the QVT standard. More precisely, we focus on the relational part of QVT.

As we have motivated in Chapter 2 languages can be described using the MOF. Therefore, we present our TGG language as a MOF metamodel. Figure 5.2 depicts the package hierarchy of our approach. Our approach itself is based on the MOF. To recall MOF basically consists of the UML infrastructure and adds a number of additional constraints. Therefore, our approach which resides in the *TGGs* package imports the *Core::Constructs* package of the UML infrastructure. The *TGGs* package contains three subpackages *Packages*, *IntegrationLinkTypes*, and *Rules*. The packages *Packages* and *IntegrationLinkTypes* contain the schema part of our TGG language whereas the rule part resides in the last package *Rules*, which will be explained in the next chapter.

5.1 Package dependencies

Figure 5.1 shows the diagram of the *Packages* package. In MOF and UML packages provide means to modularize and structure models. Well modularized and structured models are more readable and maintainable. Modularization can be used to express separation of concerns as well as responsibility issues. Furthermore, modularization can be used to improve reusability of models. Usually, packages constitute *namespaces* for the elements contained in them. Namespace means that each element can be uniquely identified by its containing package and its name. Thus, each package cannot contain two or more elements that have the same name. Additionally, packages hide their contained elements from elements contained in different packages. When an element of a package wants to refer to an element of another package the elements of the referred package must be made visible for the referring package first. As depicted

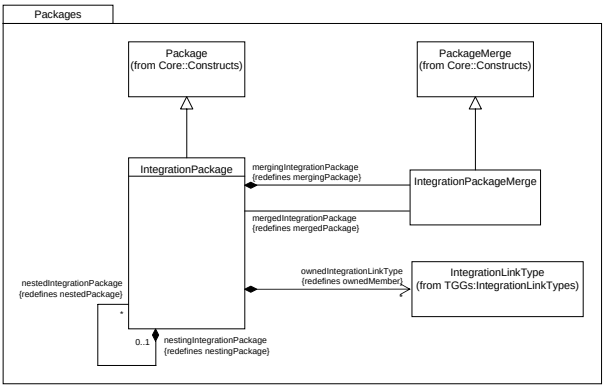

Figure 5.1: *TGGs::Packages* diagram

in Figure 5.1 `IntegrationPackages` which represent the package concept of our approach inherit from MOF package. Thus, `IntegrationPackages` can *import* arbitrary MOF packages in order to get access to their contained elements. Furthermore, `IntegrationPackages` can be nested into each other in order to form package hierarchies. `IntegrationPackages` may *merge* other `IntegrationPackages` using the same semantics as the package merge from MOF (cf. Section 2.2). Finally, `IntegrationPackages` contain `IntegrationLinkTypes` as described below.

5.2 Basic integration link type concepts

Integration link type declaration

The crucial concept of Figure 5.3 is `IntegrationLinkType`. An `IntegrationLinkType` in our approach declares a type of correspondence links. Some related TGG approaches such as [Bec08] rely on the declaration of different types of correspondence links as well. In contrast, TGG approaches such as [Wag01] rely on a single type of correspondence links only. Although, a single link type may be sufficient for some use cases the declaration of multiple link types allows for the specification of more sophisticated TGG rules. Furthermore, a dedicated

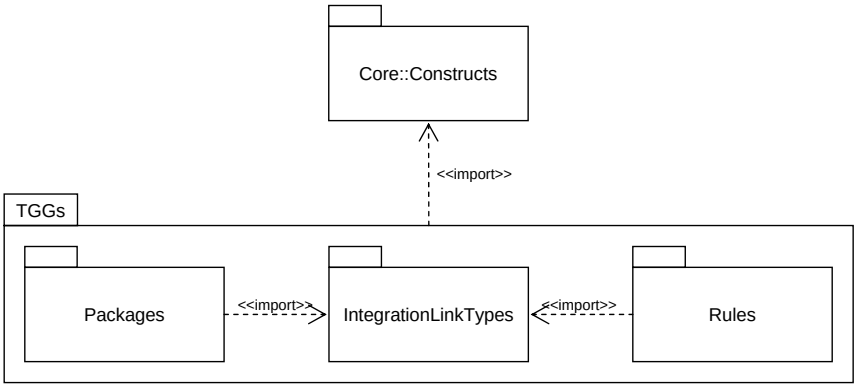

Figure 5.2: Package hierarchy of our TGG approach

link type that states that *a requirement is tested by a test case* provides more information to a user than a generic link type that only states that *one model element is linked to another model element*.

Since an `IntegrationLinkType` inherits from `MOF::Association` in our approach each `IntegrationLinkType` links two `MOF::Properties` (i.e., in our approach references to `MOF::Classes`). These `Properties` express which type of source elements may be linked with which type of target elements by links of the regarded `IntegrationLinkType`.

In our running example (cf. Figure 5.4) we declare, among others, the integration link types `PackageToSchema`, `ClassToTable`, and `SubClassToTable`. The integration link type `PackageToSchema` declares a type for correspondence links that link a `Package` from the class diagram domain with a `Schema` of the database schema domain. Similarly, the integration link types `ClassToTable` and `SubClassToTable` declare types of correspondence links that link `Classes` with `Tables`. The reason for declaring two integration link types for linking `Classes` with `Tables` is that in our approach each integration link type is the owner of at most one TGG rule (declaration). The reason for this is that we aim at compliance to the QVT standard as far as possible. TGG link types correspond to relations in QVT. Each relation in QVT has at most one domain pattern as pointed out in Section 3.3.2. In order to implement the

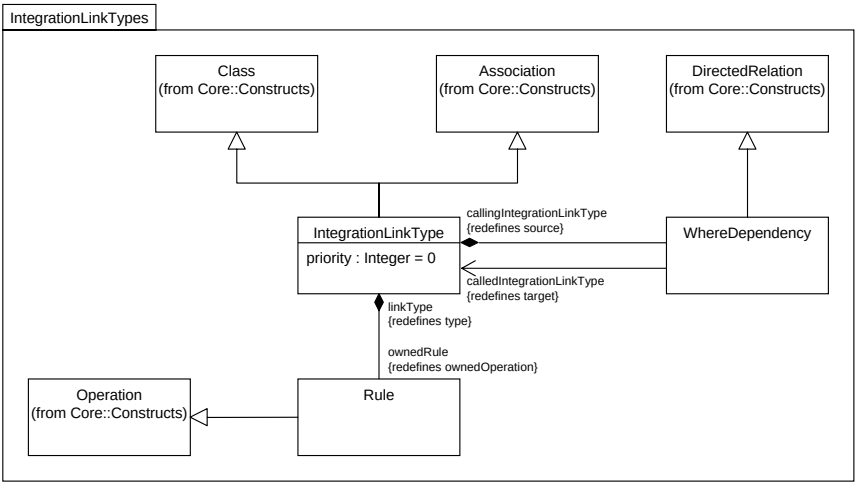

Figure 5.3: *TGGs::IntegrationLinkTypes* diagram

requirements R1 and R2 from Section 3.1 we need two TGG rules and, therefore, two integration link types.

Multiplicity constraints

Finally, the (association) ends of each `IntegrationLinkType` are provided with `Multiplicities`. These multiplicities constrain with how many model elements of the target type one model element of the source type is linked with and vice versa. Using multiplicity constraints the user can express that for instance *each requirement must be tested by at least one test case.* The other way round the user can claim that *each test case tests exactly one requirement*. Multiplicity constraints are not regarded at model integration time. Rather, these constraints are evaluated when the user wants to know whether two integrated models are consistent with each other or not by means of a completeness check.

Regarding our running example the TGG schema depicted in Figure 5.4 states that each `Class` is linked with up to one `Table` by a link of type `ClassToTable`. Furthermore, each `Table` is linked with exactly one `Class` by a link

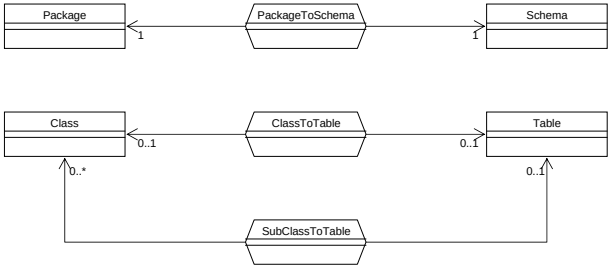

Figure 5.4: Basic concepts of a TGG schema

of type `ClassToTable`. Both constraints are implied by requirement R1 from Section 3.1. The requirement demands that only persistent classes are mapped to tables. The other way around it is obvious that each table corresponds to one (main) class. Requirement R2 demands that each persistent class which inherits from another class is mapped to the same table the parent class is mapped to. Therefore, each `Class` is linked to up to one `Table` by a link of type `SubClassToTable`. The other way around each `Table` is linked with an arbitrary number of (sub-)`Classes` by links of type `SubClassToTable`.

5.3 Sophisticated integration link type concepts

Integration link type attributes

Besides the basic integration link type concepts that are supported by most of the related TGG approaches our approach provides the following additional concepts. Since in our approach `IntegrationLinkType` inherits from `MOF::Class` each `IntegrationLinkType` owns a number of `MOF::Properties` (i.e., attributes in this case). Thus, each instance of an `IntegrationLinkType` (i.e., a particular correspondence link) can be provided with meta-information on the link (e.g., name of the user that created a link, timestamp of link creation). Furthermore, these attributes can be used to store more technical information that is incorporated by TGG rules. We will give an example in the chapter on TGG rules (cf. Chapter 6).

OCL constraints

Furthermore, `IntegrationLinkTypes` can be associated with OCL constraints. Similar to the multiplicity constraints mentioned above OCL constraints are not considered at model integration time. Rather, these constraints are checked when the user wants to test a particular correspondence link for consistency. This concept is motivated by a requirement of one of our industrial partners which stated that the condition under which two model elements should be linked with each other does not necessarily coincide with the condition under which such a link should be considered as consistent. However, that means that integrating two models with each other might result in the creation of integration links that are considered as inconsistent. This conflicts with the intention of automatically keeping two models consistent with each other. Therefore, we clarify that OCL expressions that are attached to *IntegrationLinkTypes* should be regarded as additional constraints that are not part of the actual model integration specification. Nevertheless, these OCL expressions improve the expressiveness of our approach and are requested by our industrial partners.

As an example we take a look at a model integration scenario of one of our industrial partners which deals with the integration of a requirements document kept in Doors and a corresponding test case specification kept in CTE. The condition that states that a requirement and a test case should be linked to each other is that the test case refers to the identifier of the requirement. As we will see in Chapter 6 this condition can be expressed as a regular TGG rule. However, in the integration scenario each link between a requirement and a test case should only be considered consistent if the description of the requirement additionally contains the name of the corresponding use case.

That means that during model integration each test case will be linked to the requirement with the matching identifier. If there is no such requirement the test case simply will not be linked. After the creation of all integration links each integration link will be checked whether the attached requirement contains the name of the test case or not. If the regarded requirement does not contain the name of the linked test case the integration link will be considered inconsistent.

Generalization on link integration types

As we have explained above we need two TGG rules in order to map a class to a table in our running example. One rule deals with the mapping of a persistent

class which does not inherit from another class. The other rule deals with the mapping of a persistent class which does inherit from another class. Actually, this is not sufficient because we are lacking a rule that deals with the mapping of a persistent class which inherits from a persistent class which in turn inherits from a third class. The introduction of an additional link integration type that owns the third rule does not really solve this problem. Doing so would force us to come up with a fourth rule and so on. Since we want to attach each integration link type with at most one TGG rule as explained above we cannot attach the third rule to one of the already existing link integration types. The solution we have in mind is that the link integration type that deals with the integration of a persistent class which inherits from another class inherits from the first link integration type. Doing so we can come up with a TGG rule that is applicable to any persistent class which inherits from another class regardless whether this class itself inherits from another class or not. The reason is that a rule which applies to a more abstract link integration type also applies to link integration types which inherit from the abstract link integration type. For an example we again refer the reader to Chapter 6.

However, as we pointed out in [KKS07] in detail it is not possible to let an integration link type inherit from arbitrary other link integration types.

Priorities of link integration types

While performing a particular model integration task the situation can arise that multiple TGG rules (more precisely the operational rules derived from the TGG rules as described in Chapter 7) are applicable. Usually it is not intended that all rules which are applicable actually are applied. Another possibility would be to arbitrarily choose one of the applicable rules and disregard the others. Normally, this is not intended, too. Finally, the user can be asked to select one of the applicable rules. In large scale model integration scenarios the number of such user interactions might be quite large and should, therefore, be avoided.

In order to avoid the situation that more than one rule might be applicable the user would be facing the task of writing TGG rules that ensure that only one of them would ever be applicable at a time. An algorithm which decides whether a given set of TGG rules has this property is out of scope for this work. Usually users want to use so-called *Negative application conditions (NACs)* in order to ensure this property. As we will explain in Section 7.3 NACs cause serious

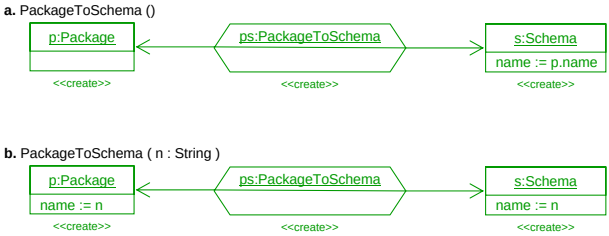

Figure 5.5: TGG rule **a.** without and **b.** with parameter

problems in the context of TGGs. Therefore, our approach does not support them at all. Rather, we provide the concept of *priorities*. Each integration link type is assigned with a priority. In the situation where more than one rule is applicable only the rule with the highest priority actually is applied. If there are more applicable rules that have the same highest priority we apply all of them. This behavior conflicts with the property of graph grammars that each element is created by one graph grammar rule only (cf. Section 7.2). We intentionally disregard this property as the possibility for applying multiple rules at the same time increases the expressiveness of our approach. Users that dislike this possibility can easily avoid it by assigning different priorities to rules which might conflict (i.e., rules that deal with the integration of model elements of the same type) during rule application.

Parametrized rule declaration

Normally, approaches that are based on the initial TGG approach from 1994 (cf. Section 4.5) do not allow for parametrized TGG rules. From a formal point of view these approaches run into problems when a rule is supposed to express that the value of an attribute of a model element from one model should equal the value of an attribute of a model element of the other model. In order to express this situation these approaches come up with rules like depicted in Figure 5.5a[1]. The problem with such a rule is that the depicted constraint accesses the value of an attribute of a model element that is just about to be created. That means that the

[1] The reader is advised to regard the depicted rules as simple graph rewriting rules as presented in Chapter 4.

desired attribute value is not available and the presented rule must be considered incorrect.

Thus, the desired constraint should be expressed as shown in Figure 5.5b. Rather than accessing the value of an attribute of a model element the value is provided as a parameter of the TGG rule. The desired constraint is implicitly expressed as the value of both considered attributes is set to the value of the parameter. Therefore, the values of both attributes are equal.

As we will explain in Section 7.1 parameters that are used in the way presented above will be resolved during the derivation of operational rules. Thus, the operational rules do not carry these parameters anymore which otherwise must be provided with actual values at rule application time by the user. Furthermore, the user may intentionally choose to use parameters that cannot be resolved at rule derivation time. The consequence is that the derived operational rules still carry the unresolved parameters which must be provided with actual values at rule application time by the user as mentioned above.

where-dependencies

Up to now the specificator has no means for explicitly manipulating the order in which the specified rules will be applied at rule application time. On the one hand this is intended as TGGs are meant to be declarative. On the other hand sometimes this can be quite inconvenient and results in specifications that are hard to understand which is surely not intended. Moreover, most TGG rules require to match a context which easily could have been provided by a previous integration step. This results in an undesired matching overhead with unnecessary high costs at runtime. For instance, the application of the rule depicted in Figure 5.6a requires for each class the matching of the package, the integration link, and the schema. Usually the TGG rules are organized in such a way that the package, the integration link, and the schema will be handled first. After that the content of the package (i.e., the classes) will be dealt with. That means that for each of these classes the context remains the same and should not be matched every time over and over again.

On a first glance this seems to be just a technical issue. Nevertheless, we adopt the concept of *where-dependencies* from the QVT standard. To recall in QVT a *where* pattern can be used in order to invoke a subsequent relation from the current relation. Thereby, the invoking relation may pass model elements to the invoked

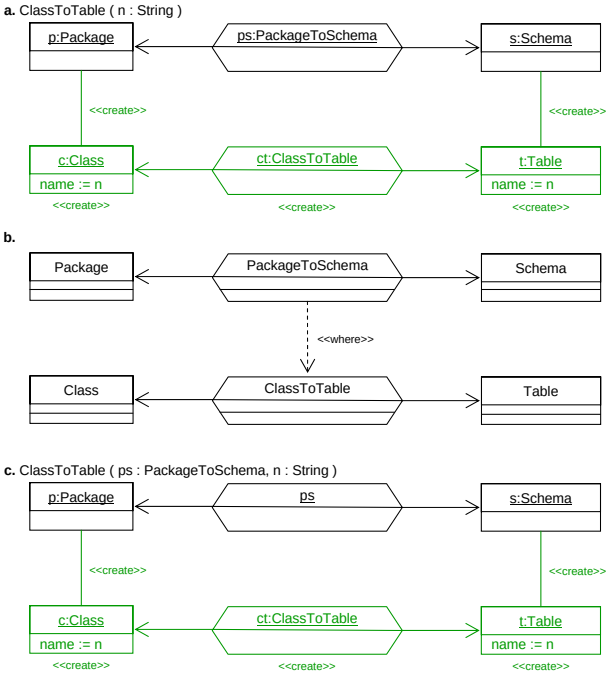

Figure 5.6: Examples of **a.** a TGG rule without provided context, **b.** the declaration of a *where*-dependency, and **c.** a TGG rule with provided context

relation. Similarly, we allow a TGG rule to invoke another TGG rule as illustrated in Figure 5.6b. In accordance to QVT we call a TGG rule that is not invoked by another TGG rule a *top-level* rule, whereas the remaining rules are called *non-top-level* rules. In contrast to the QVT standard we do not allow the invoking TGG rule to pass arbitrary model elements to the invoked TGG rule. Rather, we assume that the invoking TGG rule passes the just created integration link to the invoked TGG rule as this is the most useful model element (cf. Figure 5.6c). Thus, our concept of *where*-dependency is slightly less expressive than QVT's *where*-concept. However, concerning the example of dealing with packages and contained classes the more operational concept of *where*-dependencies appears to be more intuitive and user-friendly. Therefore, invoking TGG rules from other

QVT Relational	TGGs
Transformation	Outermost integration package
Not explicitly discussed	Modularization concepts
Typed models (EMOF)	Source and target models (CMOF)
Predicates	OCL expressions
Functions	*Intentionally not supported*
Relation	Integration Link Type
Relational domain	Source or target model
Where-clause	*Partially supported*
Keys	*Intentionally not supported, yet*
Not supported	Priorities
Simulated support	Multiplicities
Simulated support	OCL at schema level
Not explicitly discussed	Inheritance for Link Integration Types
Not explicitly discussed	Attributes for Integration Links
Parametrized relations	Parametrized Link Integration Types

Figure 5.7: Comparison of QVT Relational and TGGs

TGG rules is not a technical issue only it is also a matter of usability. As we will explain in Section 7.1.4 *where*-dependencies have impact on the operational rules at rule derivation time.

5.4 Mapping to QVT Relational

When TGGs were proposed in 1994 and even when we started our approach in 2002 the QVT specification as it looks now has not been published. Therefore, the metamodel of our approach and the metamodel of QVT are not the same. Fur-

thermore, since the current metamodel of QVT suffers from a number of defects as pointed out in Section 3.4 we chose not to switch to QVT's metamodel, yet.

However, the QVT standard looks very similar to our own approach. As the QVT standard aims at being the most important and accepted model integration standard in industry we want to informally provide a mapping of concepts from our approach to concepts from the QVT standard. By means of these mappings we claim that our approach provides some sort of implementation of the QVT standard based on the well-known formalism of graph grammars. Furthermore, we can use these mappings to replace the metamodel of our approach to the metamodel of the QVT standard when its defects have been fixed. We focus on the QVT relational package (and the underlying base package) as it corresponds to our TGG approach and is meant to be equally expressive as the QVT core package.

First of all, a *transformation* in QVT corresponds to the *outermost integration package* in our TGG approach. Our TGG approach provides means for the modularization of model integration specifications (i.e., package dependencies). QVT inherits such concept from MOF but does not explicitly discuss the semantics and the usage of these concepts. Therefore, we have to assume that these concepts more or less are accidentally included in QVT. *Typed models* in QVT correspond to typed (source and target) *models* in our approach. QVT's *predicates* correspond to OCL constraints that are attached to integration rules in our approach. *Functions* which represent queries in QVT have no counterpart in our TGG approach. Rather, a number of queries will be derived from each declarative TGG rule. Additional queries cannot be considered as part of a declarative model integration specification and, thus, have to be specified operationally. Each *relation* in a QVT specification corresponds to a *integration link type* in our approach whereas each *relational domain* corresponds either to the *source* or the *target model* in our TGG approach. As already explained above a subset of possible *where* expressions in QVT can also be specified in our TGG approach. Finally, as the concept of *keys* is unsatisfactory in QVT there is no corresponding counterpart in our TGG approach, yet (cf. Section 3.4).

The concepts *priorities*, *multiplicities*, and *OCL expressions at schema level* in our TGG approach have no explicit counterparts in QVT. Nevertheless, *multiplicities* and *OCL expressions at schema level* can be represented as *functions* in QVT. To the best of our knowledge *priorities* cannot be simulated in QVT at all. Rather, QVT relies on *Negative Application Conditions* (cf. Section 7.3) at rule level. Our TGG approach supports inheritance on link integration types and, thus,

on link integration rules. Furthermore, our approach allows for the specification of attributes for link integration types. Again, QVT inherits both features from MOF but does not discuss their semantics for relations. Finally, QVT as well as our TGG approach supports the parametrization of relations and link integration types, respectively. Figure 5.7 summarizes the correspondences and differences between QVT and our TGG approach.

6 TGG rule language

Besides a triple graph schema as presented in the preceding chapter a triple graph grammar is composed of a set of triple graph grammar rules. Again we describe basic and sophisticated elements that are supported by our approach and provide a mapping to the QVT (relational) standard.

Conceptually, a TGG rule describes the simultaneous modification of source, target, and integration model. The set of all TGG rules of a TGG specification, therefore, describes the simultaneous evolution of source, target, and integration model. As the simultaneous evolution of all regarded models is seldom useful in practice a number of operational rules which can be applied in order to realize various integration scenarios will be derived from each TGG rule as described in Chapter 7.

6.1 Basic elements

Figure 6.1 presents the package containing the metamodel of TGG rules in our approach. Basically, each declaration of a TGG `Rule` (cf. Chapter 5) is provided with a `RulePattern` which constitutes the body of the rule. Each `RulePattern` consists of a number of `RuleElements`. `RuleElements` either are `Objects`, `Links`, or `IntegrationLinks`. `RuleElements` that are marked as `created` by setting the value of `isCreated` to `true` are meant to be created during rule application. In terms of graph rewriting these elements belong to the right-hand side of the rule. `RuleElements` that are not marked as `created` (i.e., `isCreated == false`) are meant to be matched during rule application. In terms of graph rewriting these elements belong to the left-hand side of the rule. To recall when one or more elements of the left-hand side of a rule cannot be matched during rule application the application of the rule as a whole fails and no modifications are committed. `Objects` and `Links` are matched or created either in the source or the target model, whereas `IntegrationLinks` reside in the integration model.

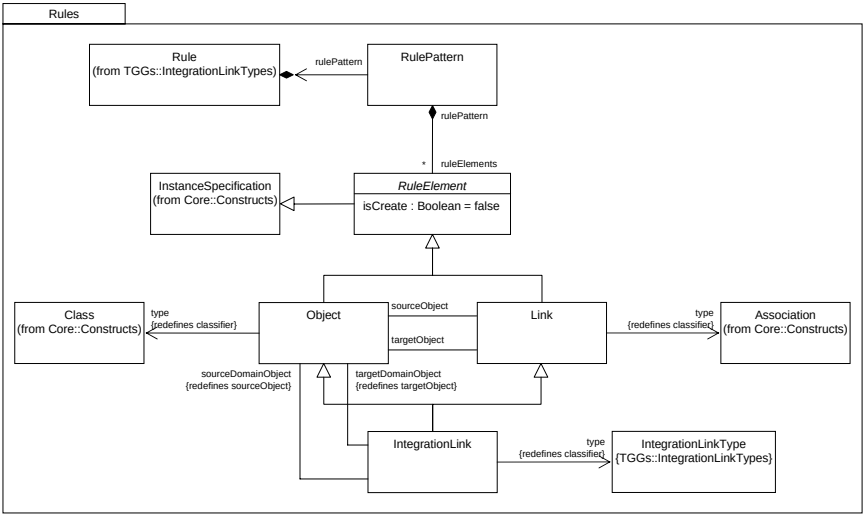

Figure 6.1: *TGGs::Rules* diagram

Basically, matching an `Object` means that the depicted `Object` is bound to an element of the appropriate type in the regarded model. Attached `Links` and linked `Objects`, called the context of this `Object`, restrict the number of possible matches. Matching a `Link` means that the depicted `Link` is bound to a link of the appropriate type in the regarded model. Furthermore, the `Objects` attached to the `Link` must also be bound and linked to each other by the regarded link. Creating `Objects` and `Links` means that new elements of the provided type are added to the regarded model and attached to any provided context.

As `RuleElements` inherit from `InstanceSpecification` each `RuleElement` can be provided with an arbitrary number of `Slots`. A `Slot` contains a value specification. A value specification can be used to restrict the number of candidates for a match by demanding that an attribute of a candidate has the specified value. Naturally, this kind of value specification can only be used for elements of the left-hand side of a rule. Furthermore, a value specification can be used to initialize the value of an attribute of an element of the right-hand side of a rule.

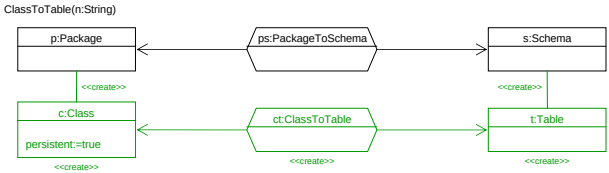

Figure 6.2: TGG rule with a simple value specification

Figure 6.2 depicts a TGG rule with a simple value specification. According to the requirements of our running example as pointed out in Section 3.1 only `persistent Classes` are transformed into a corresponding `Table`. Therefore, the value of the `persistent` attribute of the new `Class` is set to `true` by the given value specification.

Besides simple value specifications, whose right-hand sides are simple values, our approach supports two additional constructs. First of all, the right-hand side of a value specification may be an arbitrary expression which has to be evaluated at rule application time. Secondly, such an attribute expression may refer to parameters declared in the rule declaration as presented in Section 5.3.

Figure 6.3 illustrates the more complex value specification expressions. First of all, the value of the `name` attributes of the objects `a` and `col` depend on the parameter `n` which is declared in the signature of the rule. Secondly, the value of the attribute `name` of `col` has to be calculated according to the given expression. Thereby, we realize the requirement of our running example given in Section 3.1 that the names of some `Columns` must be provided with prefixes[1].

6.2 Sophisticated elements

Currently, our TGG approach does not provide any additional sophisticated rule elements. However, there are a number of imaginable extensions. Although our approach does not implement these extensions yet, we want to discuss some of our ideas now.

[1] In fact we provide the names of all columns with prefixes but the prefixes initially are blank.

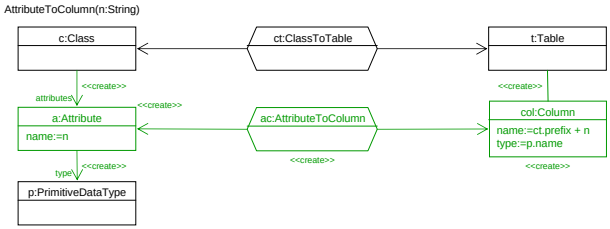

Figure 6.3: TGG rule with complex value specification

First of all we want to emphasize that our approach intentionally does not support *Negative Application Conditions* (NACs). As we will explain in Section 7.3 NACs cause presumably irresolvable problems at rule derivation time. Be advised that our approach provides a powerful replacement for NACs based on priorities as explained in Section 5.3. So far we see no way for adopting NACs into our approach in future.

Furthermore, we intentionally do not allow for marking TGG rule elements with the «`destroy`» keyword as presented in Section 4.4. In contrast to NACs the «`destroy`» keyword does not cause problems at rule derivation time. Any rule elements that are marked with «`destroy`» in a declarative TGG rule would also be marked with «`destroy`» in the resulting derived rules. Graph grammars in contrast to graph rewriting systems are meant to construct a graph just by adding new (terminal) elements. The reason is that the deletion of (terminal) graph elements in a graph grammar rule would make it very hard if not impossible to parse an existing graph. Parsing a graph means to calculate a sequence of rule applications that results in the regarded graph.

For instance in the context of TGGs forward transformation conceptually means to parse the source graph in order to calculate the sequence of rule applications[2] that resulted in the source graph. The target graph can then be created by applying the same sequence of corresponding rules[3]. The same applies more or less for the other integration scenarios as well. Therefore, being able to (efficiently) parse a graph is essential for TGGs. In order to achieve this we intentionally disallow for the deletion of model elements in declarative TGG rule

[2]This refers to the source parts of TGG rules.
[3]This refers to the target parts of TGG rules.

specifications. Again, we do not plan to add the feature of model element deletion to our approach in the future[4].

Some graph grammar-based approaches provide the concept of *optional rule elements*. Thereby, *optional* means that the considered rule does not fail when an optional rule element cannot be bound. However, when an optional rule element can be bound it can be modified by the considered rule (e.g. it can be deleted, attached to other rule elements, or its attribute values can be changed). In fact optional rule elements are just an abbreviation for more complex rules which check whether the optional elements can be bound and modifies them.

This concept can easily be adopted to TGGs as it is. Generally, optional rule elements themselves are only allowed on the left-hand side of a rule. However, optional rule elements are only useful when they are subject to modifications. Therefore, for rule derivation purposes it has to be analyzed what happens to optional rule elements and their attached modifications when deriving an operation rule from the considered TGG rule. This is future work.

A similar discussion concerns the concept of *set nodes*. A *set node* is bound to the set of all found matches of the given node specification. It allows for the modification of all nodes in the calculated set of matches. Again, it has to be analyzed what happens to set nodes while the derivation of operational rules. The concepts of *optional* and *set* become even more sophisticated when they are applied to subgraphs rather than single nodes. Again, this is future work.

The last concept we want to propose is the concept of *optionally created rule elements*. Optionally created rule elements should only be created when they cannot be bound at rule application time. This ensures that an optionally created rule element is created only ones when applying the regarded rule and reused in subsequent applications of this rule. Again, this concept is only an abbreviation for a more complex rule which checks whether the given rule element can be bound and creates it otherwise. This concept is particularly useful when a considered rule contains multiple optionally created rule elements.

For rule derivation purposes each rule element that is marked as *optionally created* and has to be moved to the left-hand side of the derived rule will be transfered and is marked as a normal rule element. The reason is that after the application of the regarded declarative rule it is guaranteed that the considered rule element

[4]Observe that QVT also does not support the deletion of model elements.

QVT Relational	TGGs
Rule	Rule
Patterns (except *where*-pattern)	RulePattern
TemplateExpression	RuleElement
PropertyTemplateItem	Value Specification

Figure 6.4: Comparison of QVT Relational and TGGs *(cont.)*

exists. Therefore, the rule element has to be matchable when applying the corresponding operational rule. Again, this is future work.

6.3 Mapping to QVT Relational

Even on the level of rules QVT and our TGGs look quite similar to each other. The term *Rule* from our approach corresponds to the term *Rule* from QVT. In QVT each rule consists of a number of *Patterns* (e.g., *when-* and *where-patterns*, *domain patterns*). The set of *Patterns* other than the *where-pattern* of a QVT rule corresponds to the *RulePattern* of a TGG rule. Thereby, the term *TemplateExpression* from QVT corresponds to the term *RuleElement* from our TGG approach. *TemplateExpressions* that have not already been bound beforehand (e.g. by a *when-pattern*) correspond to the *RuleElements* on the right-hand side of the regarded TGG rule, whereas already bound *TemplateExpressions* correspond to the *RuleElements* on the left-hand side. Furthermore, *PropertyTemplateItems* that are attached to *Template Expressions*[5] correspond to *Value Specifications* in our approach. Figure 6.4 summarizes these correspondences.

Figure 6.5 depicts a TGG rule and the corresponding QVT rule in order to illustrate the similarities of both approaches. The TGG rule from Figure 6.5a. means that for each persistent class that is added to an already existing package a corresponding table will be added to an already existing schema which corresponds to the regarded package and vice versa. Thereby, the names of the class and the

[5]More precisely: ObjectTemplateExpressions.

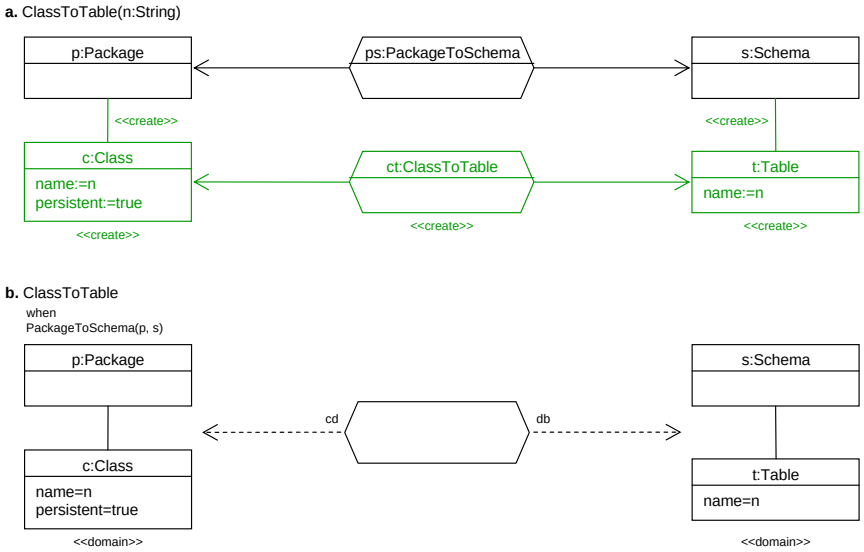

Figure 6.5: Comparison of **a.** a TGG rule and **b.** the corresponding QVT rule

table are identical. Similarly, the QVT rule from Figure 6.5b. means that a class is related to a table when the package to which the class belongs is related to a schema that contains the table. Furthermore, the names of the class and the table must be the same.

Both depicted rules are meant to be declarative and, therefore, can be applied in order to ensure the consistency between a class and its corresponding table. If there is no corresponding table for a class such a table will be created and vice versa. Thus, the TGG rule and the QVT do not only look quite similar to each other; they basically have the same semantics. Generally speaking the QVT standard and our TGG approach are intentionally nearly identical.

The major benefit of TGGs is that they rely on a proper formal foundation, i.e. theory of graph grammars. In contrast the semantics of QVT is more or less only informally given. On the one hand TGGs can be utilized as the missing formal foundation for the QVT standard. On the other hand the implementation of our

TGG approach can, therefore, be regarded as one of the first implementations of the declarative part of QVT.

7 Operational rules

In this chapter we present how we automatically derive operational graph tranformations from declarative triple graph grammars as presented in the preceding chapters. We want to emphasize that the rule derivation strategies as presented in this chapter can be regarded as a proposal for rules that are useful for our model integration approach. Nevertheless, we do not claim that the set of derived rules is complete. The reader may come up with own additional rules if they feel need to. Furthermore, we illustrate in which way these operational graph transformations can be applied in order to realize the desired integration of models.

7.1 Derivation strategies

Basically, a TGG describes the simultaneous evolution of two models and their relationship by means of correspondence links. In practice the simultaneous evolution is seldom useful. The application of a TGG as it is would mean that while a user is modifying one model by using a tool he is simultaneously modifying another model residing in another tool which might be used by a different user and vice versa. That means that a user of one tool is constantly bothering other users with updates of their models. Rather, each user wants to work with their tools as usual and synchronize with changes done by other users only at certain points of time.

Such a synchronization needs the identification of changes to models since the last synchronization. The identified changes must then be applied to related models correspondingly. The proper application of changes requires appropriate operations which ensure that the considered models are consistent with each other after synchronization. The crucial point of TGGs is that the set of needed operations can automatically be derived from the declarative rules.

7.1.1 Classical rules

The initial TGG approach from 1994 [Sch94] identified the following three integration tasks which should be covered by the derivation of operational graph rewriting rules from the declarative TGG rules.

1. **Model transformation.** This integration task arises when one model has got new elements that should be related to elements in the other model but the other model does not contain these corresponding elements yet and vice versa. Therefore, corresponding elements have to be added to the second model and have to be related to the new elements of the first model by means of correspondence links.
2. **Consistency checking.** This integration task arises when the user wants to check whether two models are consistent to each other. If the models are inconsistent the user wants to get a report of all inconsistent model elements. To this end all existing correspondence links have to be checked whether they are still valid. Furthermore, each model has to be checked whether it contains model elements that should correspond to other model elements but do not yet.
3. **Correspondence link creation.** This integration task arises when the user has got two models that are not provided with any correspondence information yet and wants to calculate the missing correspondence information by means of correspondence links.

7.1.2 Additional rules

In addition to the classical rules we have identified the following integration tasks.

4. **Attribute value propagation.** This integration task arises when two elements are consistent to each other by means of their structural properties but their attribute values do not match (i.e., the graph pattern of the TGG rule can be matched but some of the attribute conditions are violated). Some of the attribute values of the corresponding elements have to be adjusted in order to recover consistency.
5. **Element deletion propagation.** This integration task arises when a correspondence link has a dangling end. This can only occur if the user has

intentionally deleted the element that was attached to the regarded link beforehand. In order to recover consistency the corresponding link as well as the remaining corresponding element have to be deleted as well since that reflects the intention of the user.

6. **Correspondence link deletion.** This integration task arises when two elements are related by a correspondence link but should not be linked anymore. The solution is to just delete the undesired correspondence link.

7.1.3 Operational rule derivation

Using the declarative TGG rule from Fig. 6.5a. of our running example we explain the derivation of our operational rules. The derivation of an operational rule from a declarative rule consists of two steps. Firstly, we have to process the pattern of the declarative rule and to deal with the contained attribute assertions. Secondly, we have to resolve any parameters the declarative rule might have such that the resulting operational rule is unparameterized. If we cannot eliminate all parameters the resulting rule requires user interaction at application time asking the user for the actual value of each remaining parameter. Basically, the pattern of an operational rule is the same as the pattern of the corresponding declarative rule. The operational rule is derived from the declarative rule by moving some elements of the pattern from the right-hand side of the rule (i.e., elements that are created by rule application) to the left-hand side of the rule (i.e., elements that are to be matched at rule application time).

Derivation of model transformation rules

In order to support model transformation we derive two operational rules from a given declarative rule. One rule is called *forward transformation rule*, the other is called *backward transformation rule*. Forward transformation is a task that transforms one model into another model in the "natural" direction of the transformation. Backward transformation performs this task the other way round. It is up to the user to define which direction is considered to be natural. For instance it can be considered natural if one model is from an earlier development phase than the other and the transformation is performed in this direction, it can be considered natural if one model is more abstract than the other and the transformation is performed in this direction, and so on. We call the model that is transformed into the other by transformation rules the *source model*. We call the other model the *target model*.

In order to derive the forward transformation rule from a TGG rule we move all elements that refer to the source model that currently belong to the right-hand side of the rule to the left-hand side of the rule. Since elements of the source model that are marked as *optional create* in the declarative rule would be created while rule application if they do not already exist these elements generally[1] become plain obligate elements during rule derivation. This means that elements from the source model that would have been created by the application of the declarative rule now have to be matched in the source model in order to apply the resulting forward transformation rule. Only elements from the target model and the correspondence model will be created now. This results in the desired model transformation. Furthermore, we replace any attribute assignment of elements of the source model by corresponding attribute conditions. This means that the attribute values of elements of the source model are no longer modified. Rather, the attributes are now checked for certain values. This corresponds to moving elements from the right-hand side of a pattern to the left-hand side. Finally, we resolve any parameter of the declarative rule if possible as follows. We consider the occurrence of a parameter in the source model as the definition of the parameter. We consider the occurrence of the same parameter in the target model as the usage of the parameter. Therefore, we replace any usage of a parameter by its definition.

Fig. 7.1 illustrates the derivation process. The derived forward transformation rule takes the to be transformed `Class c` as input. From `c` the rule navigates over the `Package p` and the correspondence link `ps` to the `Schema s` to which a new `Table t` should be added to. The rule then creates `t` and a correspondence link `ct` in order to link `c` and `t` with each other. Furthermore, by resolving the parameter `n` of the considered TGG rule the value of the `name` attribute of `t` is set to the `name` attribute of `c`. Correspondingly, we derive the backward transformation rule from the considered TGG rule by switching the roles of source and target model.

Derivation of consistency checking rules

An operational rule that checks a given correspondence link whether it still is valid does neither create new elements nor modify existing ones. Therefore, we have to move all elements from the right-hand side of the pattern of the declarative rule to the left-hand side of the pattern in order to derive the desired operational rule. Observe that the consistency of the given correspondence link intentionally

[1] This applies to all rule derivation strategies.

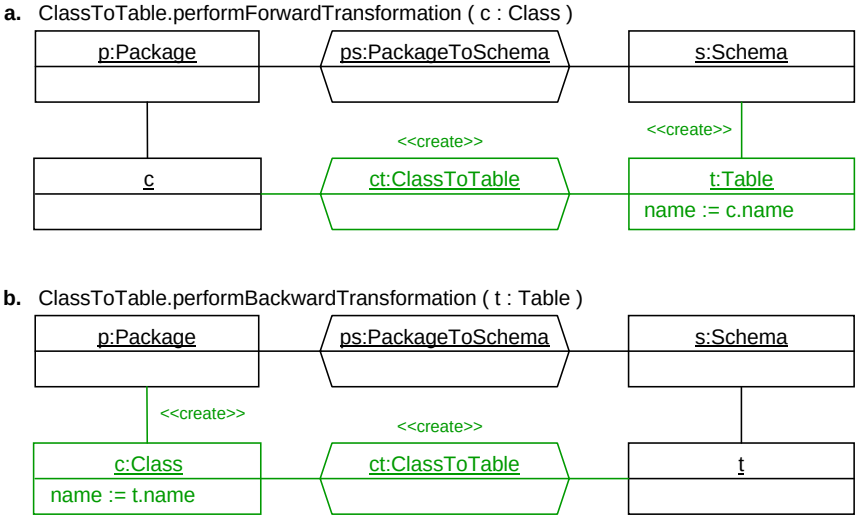

Figure 7.1: Derived model transformation rules

does not depend on the consistency of correspondence links of the left-hand side of the declarative rule. Otherwise, it would be unclear whether a correspondence link is inconsistent because its rule pattern is violated, because other correspondence links are inconsistent, or both.

Furthermore, we also have to replace any attribute assignment by a corresponding attribute condition. For resolving parameters of the TGG rule we can arbitrarily choose one occurrence of the regarded parameter as its definition and the other as usages. Again we replace all usages of a parameter by its definition as illustrated in Fig. 7.2. The resulting rule now checks whether the pattern that has been valid at creation time of the correspondence link still can be matched (i.e., the correspondence link still is valid).

Derivation of correspondence link creation rules

For supporting link creation we derive two operational rules from a given TGG rule. We can link one element of one model with elements of the other model and vice versa. Therefore, we have a *forward correspondence link creation rule* and a *backward correspondence link creation rule*. The rules only differ in the element

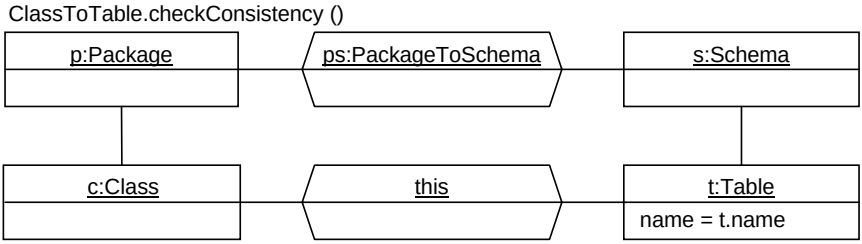

Figure 7.2: Derived consistency checking rule

from which the pattern matching of the regarded rule starts. In order to derive the desired rules we move all elements that refer to elements of the to be integrated models from the right-hand side to the left-hand side of the pattern. The only element that resides at the right-hand side is the to be created correspondence link. Additionally, we replace all attribute assignments that are not owned by the to be created correspondence link by attribute conditions. For resolving parameters of the TGG rule we can arbitrarily choose one occurrence except the to be created correspondence link of the regarded parameter as its definition and the other as usages. Again we replace all usages of a parameter by its definition.

The derived rules are illustrated by Fig. 7.3. The forward link creation rule starts at `Class c`. By navigating over `Package p`, the correspondence link `ps`, and `Schema s` the rule tries to find a `Table t` such that the given attribute condition holds. If such a `Table` can be matched the rule links `c` with `t` by a new correspondence link `ct`. Correspondingly, we derive the backward link creation rule from the considered TGG rule by switching the roles of source and target model.

Derivation of attribute value propagation rules

In order to propagate changes of attribute values we derive two operational rules from a given TGG rule. Again, such a propagation can be perform in forward and in backward direction. Therefore, we derive a *forward attribute value propagation* and a *backward attribute value propagation* rule. For deriving the forward attribute value propagation rule we move all elements of the right-hand side of the TGG rule to the left-hand side of the resulting operational except for attribute assignments of target model elements that depend on the considered source

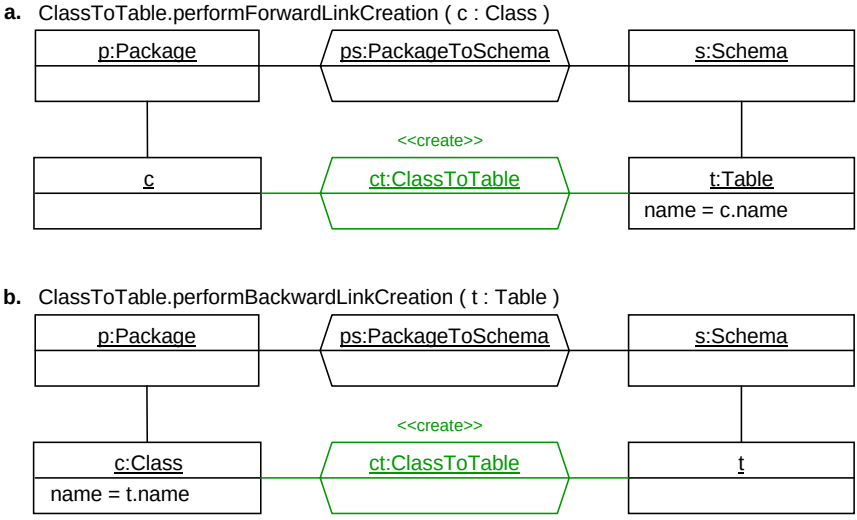

Figure 7.3: Derived link creation rules

element. Thereby, we resolve any parameters of the declarative rule as usual. Correspondingly, the backward value propagation rule is derived by just keeping the attribute assignments of source model elements that depend on the considered target element.

The derivation is illustrated by Figure 7.4. Starting from a given correspondence link the pattern navigates to the linked `Class c` and `Table t` and updates the value of the `name` attribute of `t` in case of the forward value propagation rule accordingly. The reason for additionally matching the elements `p`, `ps`, and `s` rather than only `c` and `t` is that in the general case any other elements of the pattern might have attribute assignments that are to be processed.

Derivation of element deletion propagation rules

We support the propagation of deletion of elements by two operational (i.e., *forward deletion propagation* and *backward deletion propagation*) rules derived from a given TGG rule. In case of the forward deletion propagation rule we move the source element from the right-hand side of the declarative rule to the left-hand

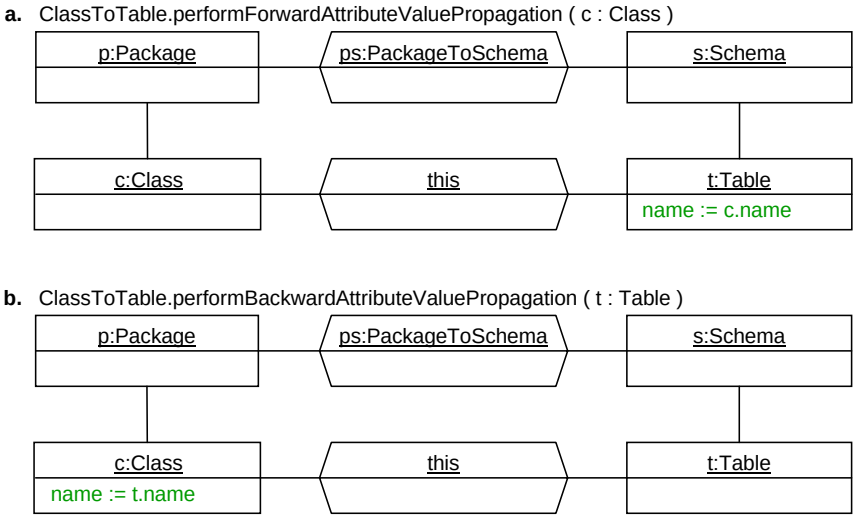

Figure 7.4: Derived attribute value propagation rules

side of the operational rule. Furthermore, we attach a negative application condition to this element. This means that it must be impossible to find a match for this element expressing that this element does not exist any more. Additionally, we attach deletion flags to all elements of the right-hand side of the declarative rule which belong to the target model and to the correspondence link as well. Finally, we transfer the context elements of the to be deleted elements in order to properly delete all links to the deleted elements. The backward deletion propagation rule is derived correspondingly.

Figure 7.5 illustrates the derivation process. In case of the forward deletion propagation rule the pattern starts from a given correspondence link. After that the rule tries to match a `Class c`. If this does not succeed this means that the element that has been linked by the correspondence link has been deleted in the meantime and the correspondence link now has a dangling link end. Therefore, the rule deletes the correspondence link as well as the other link end which is a `Table t`.

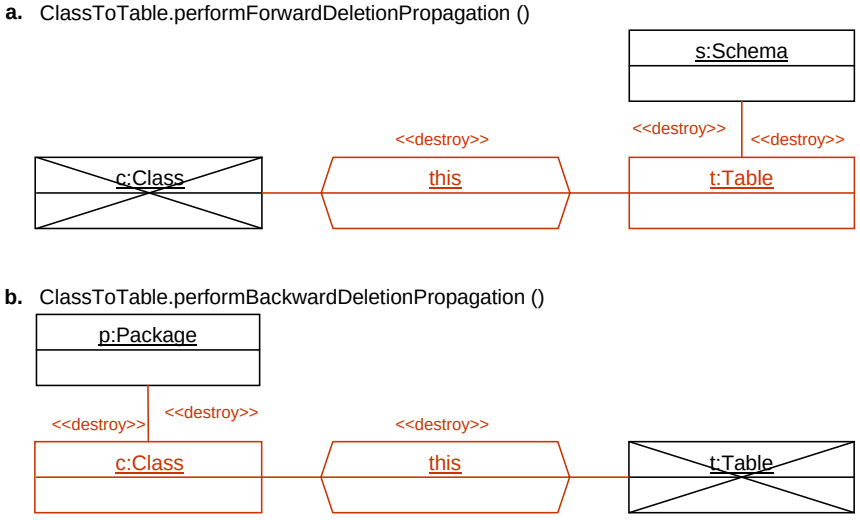

Figure 7.5: Derived element deletion propagation rules

Derivation of link deletion rules

Rules that delete existing correspondence links can be derived from TGG rules in the following way. The considered correspondence link is just attached with a deletion flag. The context elements of the to be deleted link which are the linked model elements are move from the right-hand side of the TGG rule to the left-hand side of the operational rule. These context elements are needed to properly remove the connection between the elements and the correspondence link.

This derivation is illustrated by Figure 7.6. The considered correspondence link as well as its connections to the linked elements `c` and `t` are just deleted.

We emphasizes that the set of derived operational rules as presented so far is neither mandatory nor complete. The basic idea of TGGs is just that it is very simple to derive a number of operational rules from a given TGG rule. The set of the just presented operational rules reflect only those rules we consider useful and that are used by our approach. Nevertheless, someone might drop some of the presented rules or come up with additional derivation strategies if he needs to.

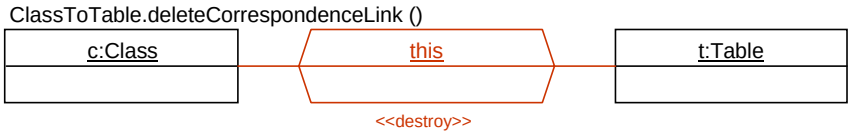

Figure 7.6: Derived link deletion rule

7.1.4 Impact of *where*-dependencies on rule derivation

As we have mentioned in Section 5.3 our TGG approach adopts the concept of *where*-dependencies from the QVT standard. To recall a *where*-dependency between a link integration type `A` and a link integration type `B` means that a rule that is responsible for an instance `a` of link integration type `A`, invokes the corresponding rule of link integration type `B`. Thereby, the invoking rule passes the just created integration link `a` as an input parameter to the invoked rule. Thus, the invoked rule does not have to match the provided integration link `a` and can easily match the model elements linked by this integration link. One aim is to save calculation time for recurrent matches of integration links, another is usability.

There are at least two possibilities of dealing with *where*-dependencies at rule derivation time. One possibility is to recursively merge the pattern of the invoked rule with the pattern of the invoking rule at rule derivation time. Although that approach is straight-forward it runs into severe problems if rules directly or indirectly invoke each other in a circular manner. Another possibility is to utilize the mechanism of graph rewriting systems as PROGRES or in our case FUJABA to specify rule invocation. This time circular dependencies are not a problem because the dependencies are not dealt with at rule derivation time. The rule invocation just stops as soon as a rule which would invoke another rule is not applicable.

7.2 Application strategies

A TGG specification consisting of a TGG schema and a set of TGG rules declaratively specifies when two given models may be considered consistent with each other. We have just described how to derive a number of operational rules that

can be derived from each TGG rule in order to maintain consistency of two given models. Now we need strategies how to apply these rules. In our approach each rule either takes one model element as input or starts from a given correspondence link. Therefore, our strategies are required to cover all model elements of the considered models and all correspondence links of the correspondence model.

The actual integration task (e.g. forward transformation, correspondence link creation, consistency checking) we want to perform determines which rules are to be applied and which models (source, target, or correspondence) are to be traversed. Additionally, the way the TGG rules are written determines in which order the regarded model has to be traversed. Usually, TGG rules are written in a top-down fashion. That means that model elements that constitute containers for other model elements are transformed prior to the contained elements. The rationale for this is that on the one hand this appear to be the most natural way and on the other hand most CASE tools require model elements to be created in already existing containers. However, if the TGG rules implicitly traverse the models in a top-down fashion the rule application strategy must accordingly traverse the considered models top-down, too. If the rule application strategy traverses the models in a different way (e.g. arbitrarily or bottom-up) the application of some rules might unintentionally fail because elements (most notably correspondence links) required by the left-hand side of a rule do not exist due to the untimely application of the considered rule.

A more intricate problem arises if the application of a rule to a given element requires elements that exist on the same containment level as the considered element. In this case the correct order in which the elements are to be traversed is not implicitly given. Therefore, the application strategy must test whether all elements that are required by a certain rule have been processed beforehand. Else, the processing of the considered element must be delayed until the required elements have been processed. If the TGG specification is erroneous this delay can result in a dead-lock which must be detected at least at rule application time.

Figure 7.7 illustrates these problems. The rule `ClassToTable.performForwardLinkCreation` contains on its left-hand side an instance of the `PackageToSchema` correspondence link type. Therefore, the application of this rule fails for `Classes c1` and `c2` if the rule `PackageToSchema.performForwardLinkCreation` has not been successfully applied to `Package p1` beforehand. Correspondingly, the application of the rule `AssocToFKey.performForwardLinkCreation` to `Association a1` fails if `Classes`

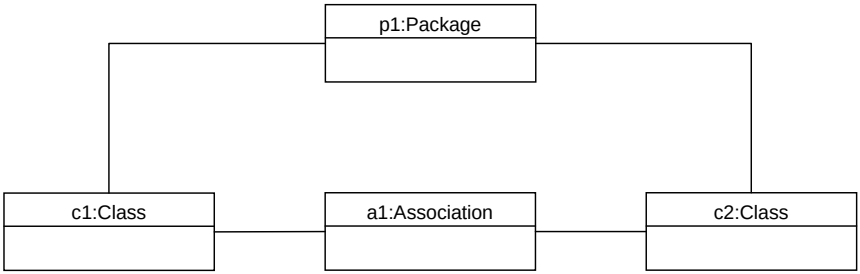

Figure 7.7: Example model for the illustration of application strategy related problems

`c1` and `c2` which both lie an the same containment level as `a1` have not been successfully processed before. Thus, processing the elements either in the following sequence `p1`, `c1`, `c2`, `a1` or `p1`, `c2`, `c1`, `a1` leads to a valid result, whereas all other sequences lead to incomplete, hence invalid results.

When processing a given model element it might be the case that more than one rule is applicable. Usually, it is not intended to apply all applicable rules. Rather, the application strategy is intended to determine one rule (in rare cases a subset of rules) that actually is to be applied. Some approaches as [Bec08] ask the user to select the desired rule. This is only feasible if the number of user interactions is small. It would be a nightmare for a user to select more than a couple of rules. The situation becomes even worse if the user has to redo this selection every time an integration task is executed. However, in large scale model integration scenarios with thousands of to be integrated model elements the number of situations where more than one rule is applicable is much too large for user interaction. To this end our approach uses priorities that are provided for each rule. The idea is that the application strategy only applies those rules which priorities are the highest. In case that there is more than one such rule we again have the choice to either apply all rules or to ask the user. Although the number of required user interactions would be noticeably lower we choose to blindly apply all remaining rules. If this is not the intended behavior it is up to the TGG specification to avoid those situations. Observe that from the formal point of view the application of more than one rule to a single model element violates the character of a (graph) grammar. Nevertheless, practice has shown that the application of multiple rules

is actually required. The author is of the firm conviction that formal requirements that would allow for proving certain theoretical properties may be violated if they constrain the whole approach from being useful in practice.

Finally, when processing a given model element it might be the case that there are multiple matches for the application of one rule. Again, some approaches such as [Bec08] ask the user to select one match from the set of possible matches that should used for rule application. As mentioned above this solution is not feasible for large scale model integration scenarios. Therefore, we choose to blindly apply the regarded rule to all possible matches. Again, it is up to the TGG specification to avoid such situations if this behavior is not intended. As already pointed out above this solution violates the character of a (graph) grammar but increases usability of our approach in practice.

Altogether, we use the following application strategy for cases where we need to traverse all model elements of either the source or the target model (i.e., model transformation and correspondence link creation). We emphasize that an application strategy heavily depends on the way TGG rules are specified. Since we use a top-down strategy as explained above the users of our approach are forced to write their TGG rules in a top-down fashion as well. Nevertheless, the original TGG approach itself does not imply a certain application strategy.

1. Start with an ordered list `L` of all outermost model elements (i.e., elements that are not contained in other elements) of the to be transformed model. Initially, this list is arbitrarily ordered.
2. For each element `e` of `L` determine the set `S` of elements that are contained by `e`.
3. Arbitrarily add `S` to the ordered list `N` of elements that are to be processed at the next iteration.
4. Process each element `e` of `L` in the following way:
   a) Determine the set `R` of operational rules that are applicable to `e`.
   b) Only keep those rules with the highest priority.
   c) For each rule `r` of `R` check whether the context of `r` requires elements that are not processed, yet.
   d) If there are such elements move `e` from `L` to the beginning of `N`.

e) Else, apply all rules of R to e. Thereby, process all possible matches for each rule.

5. If N is empty the integration task has finished.
6. If N contains all and only elements of L (i.e., all elements of L have been delayed and there are no other elements left to be processed) we have detected a dead-lock. The underlying TGG specification is considered erroneous and the integration task ends unsuccessfully.
7. Else, move all elements from N to L and continue at step 2.

In cases where we need to traverse all correspondence links (i.e., consistency checking, attribute value propagation, element deletion propagation, and correspondence link deletion) we can just traverse all correspondence links in an arbitrary order. Particularly, all of the corresponding operational rules can be applied to single correspondence links chosen by a user.

We clarify the application strategy presented above by applying it to our running example. To this end we consider the forward transformation of a class diagram as presented in Figure 7.8 into a corresponding database schema. The application strategy deals with the question in which order the elements of the source model are to be transformed and which transformation rules are to be applied.

Firstly, we have to determine the set L of all outermost elements. In our case there is only one outermost element, namely the package p. For the next iteration we have to determine the set S of elements that are directly contained in the elements of L. In our case the package p directly contains the classes c1, c2, and c3. We add the elements from S into the ordered set N in an arbitrary order (e.g. c2, c3, c1). Now we process each element from L. Therefore, we have to process package p first. We determine the set of operational forward transformation rules that are applicable to p. There is only one rule that deals with the transformation of a package, namely `PackageToSchema.performForwardTransformation(p:Package)`. As there is only this candidate rule it naturally has the highest priority. Furthermore, this rule does not contain any context elements which have to be dealt with beforehand. Thus, we just apply this rule to package p. The resulting target model looks as depicted in Figure 7.9a.

As the set N is not empty we have to continue the transformation with another iteration. We move all elements from N to L which now contains the classes c2, c3, and c1. We put all elements that are contained by any of these classes

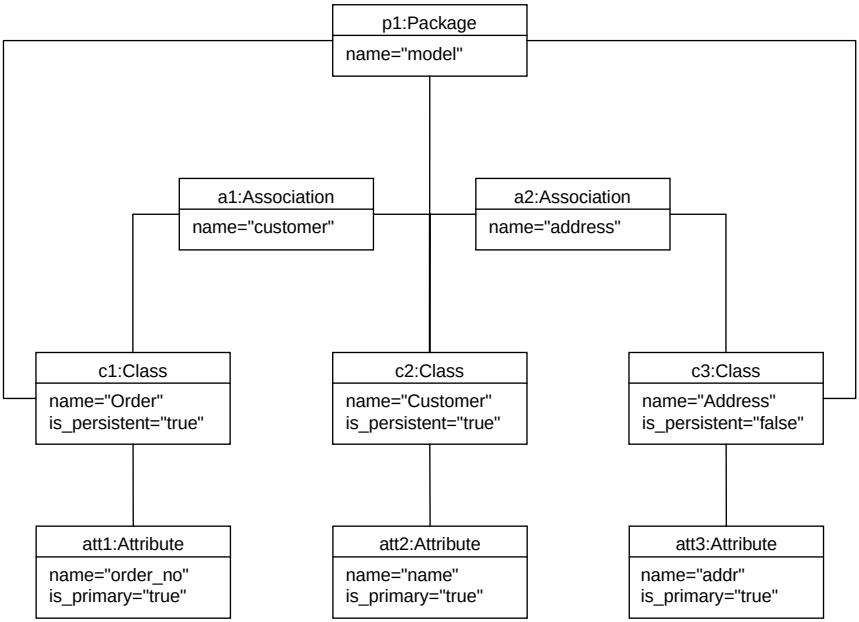

Figure 7.8: Example class diagram model

into the set `S` which contains the associations `a1` and `a2` as well as the attributes `att1`, `att2`, and `att3`. We move all elements of `S` into the ordered set `N` in an arbitrary order (e.g. `att2`, `a1`, `att3`, `att1`, `a2`). Now, the classes `c1`, `c2`, and `c3` are to be transformed. As `c3` is a non-persistent class no rule matches and `c3` is disregarded for now. The target model now looks as depicted in Figure 7.9b.

In the last iteration the remaining elements `att2`, `a1`, `att3`, `att1`, and `a2` are transformed by applying the appropriate transformation rules. Finally, the target model looks as depicted in Figure 7.9c.

In order to check whether two given models are consistent with each other or not it is insufficient to consider all correspondence links and check whether they are consistent. Besides the operational rules for checking correspondence links for consistency the TGG schema provides additional information that must be considered as mentioned in Chapter 5. On the one hand correspondence link

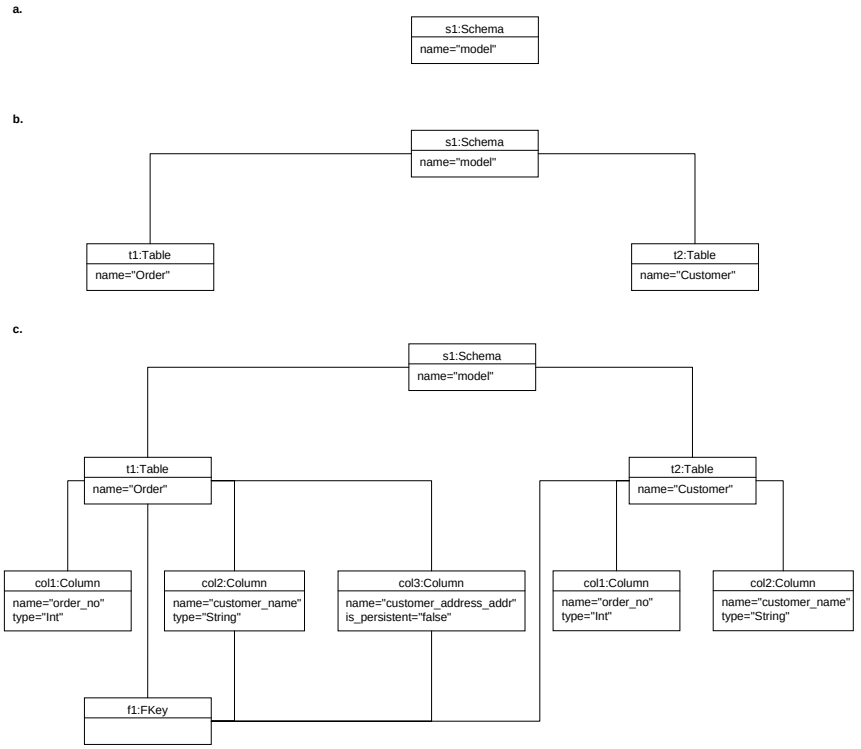

Figure 7.9: Resulting database schema model

types are provided with multiplicities. Multiplicities express how many model elements of a given type have to be linked to a given model element by correspondence links of the regarded type by means of a lower and an upper bound. On the other hand correspondence link types can be provided with additional (OCL) constraints. These constraints express situations in which an existing correspondence link is inconsistent although its underlying rule pattern is not violated. Therefore, we need a more sophisticated algorithm for checking two models for consistency as described in the following.

1. Check all already existing correspondence links for consistency. Report all inconsistent links.
2. Create new correspondence links by applying the `performForwardLinkCreation` or the `performBackwardLinkCreation` rules as explained above.
3. Check all correspondence links whether they do not violate the constraints specified in the TGG schema. Report all violating links.
4. Traverse the model elements of both models and check whether they do not violate the multiplicity constraints specified in the TGG schema. Report all violating model elements.
5. Both models are considered consistent with each other only if no links or elements have been reported.

Since most of the time two models will be inconsistent with each other we are interested in means how to recover consistency. First of all we are facing the problem that we can only identify when two models are inconsistent with each other. We cannot decide which of the models is correct and which is violating consistency. Most of the time it makes sense to assume that the model that has recently been modified is the correct one and changes to it should be propagated to the other model. However, the only one who can decide what is correct and what is not is the user. This choice can be done at two levels of granularity. On the one hand the user can choose which model as a whole should be considered correct. On the other hand the user can perform this choice for each detected inconsistency. Whilst the first possibility might be too coarse-grained the second possibility tends to be too fine-grained for large-scale model integration scenarios. Nevertheless, asking the user is the only possibility for being sure that repair actions only do what the user wants them to do.

However, depending on the type of inconsistency the application of some operational rules as repair actions can be performed at least (semi-)automatically. In case that a correspondence link has been reported as inconsistent in Step 1 one of the following reasons apply. Firstly, some of the attribute conditions of the underlying rule pattern might be violated. In this case the application of the corresponding attribute value propagation rules should fix the problem. Moreover, the matching of the structural part of a pattern might fail. This can occur when a correspondence link has a dangling link end. In this case the application of deletion propagation rules fixes the problem. Finally, when other parts of the un-

derlying rule pattern cannot be matched the application of model transformation and correspondence link creation rules should fix the problem.

When a correspondence link is identified as inconsistent in Step 3 user interaction can only be avoided if a powerful constraint solving mechanism is available in the general case. Finally, when model elements are identified as inconsistent in Step 4 there are two possibilities. When a model element violates the lower bound of a multiplicity constraint the application of model transformation rules creating the appropriate model elements in the other models as well as the required correspondence links should fix the problem. When a model element violates the upper bound of a multiplicity constraint the user is required to choose model elements that should be deleted.

7.3 On negative application conditions

As we have mentioned in Chapter 5 our approach does not support *Negative Application Conditions (NACs)*[2] as introduced in Section 4.4. The reason is that we do not know how to deal with a negative application condition during rule derivation. Basically, we have two obvious possibilities: Firstly, we could transfer a negative application condition from the declarative to an operational rule as it is. Secondly, we could just drop the negative application condition. We will now give a short counterexample which motivates that neither of both possibilities is correct. As a result we conclude that NACs cannot be dealt with and, therefore, are excluded from our approach. We emphasize that our approach incorporates a powerful replacement called *priorities* (cf. Section 5.3) that is able to cope with most situations where NACs would be needed otherwise.

Our counterexample deals with the forward transformation of a linked list into a corresponding identical linked list. Figure 7.10a depicts a metamodel for linked lists. Each `List` contains a number of `Elements`. Each `Element` can have up to one `successor`. Additionally, we demand that if the list contains at least one element exactly one of these elements has no predecessor (i.e., each list has a first element). Furthermore, we demand that if the list contains at least one element exactly one of these elements has no successor (i.e., each list has a last element). Figure 7.10b shows the to be transformed source list.

[2] While rule application a part of a host graph only matches a rule containing NACs if it is impossible to match any of the object patterns that are marked as a NAC.

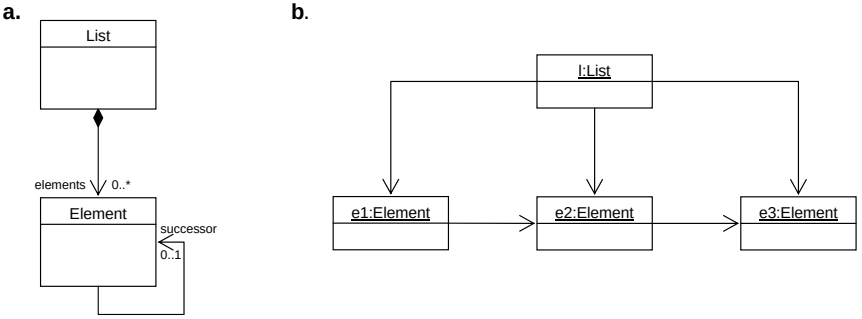

Figure 7.10: **a.** Metamodel of linked lists, **b.** To be transformed source model

Since we want to transform the given list into an identical list we can focus on the source part of the model transformation rule and can omit the trivial correspondence link and target parts. Figure 7.11a depicts the source part of the declarative model integration rules. The first rule creates a new `List`. The second rule adds a first `Element` to the `List`. The NAC ensures that the `List` has no further `Elements`. The last rule adds a successor to an existing `Element` of the `List`. The NAC ensures that only `Elements` can be provided with successors that do not already have a successor. Thus, the example from Figure 7.10b can be created by applying the first rule, then the second rule, and finally the last rule twice.

When deriving operational forward transformation rules from the declarative rules we have to deal with the NACs. Figure 7.11b depicts the source part of the forward transformation rules if we decide to keep the NACs. Correspondingly, Figure 7.11c[3] shows the forward transformation rules if we drop the NACs. In order to perform the desired forward transformation we have to transform each model element of the given source model by applying one of the given forward transformation rules as discussed in the preceding section.

We start with `List l` and apply the first forward transformation rule from Figure 7.11b. After that we have to deal with one of the `Elements` `e1`, `e2`, or `e3`. The application of the second rule is not possible as the `List` already contains three elements and the rule is only applicable if the `List` contains only one

[3] Disregard the given priorities for now.

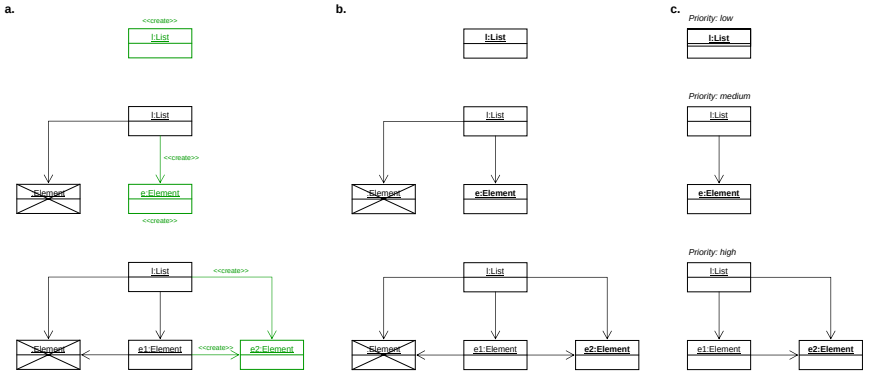

Figure 7.11: **a.** Source part of the declarative model integration rules, **b.** Derived forward transformation rule parts with NACs, **c.** Derived forward transformation rule parts with priorities

`Element`. The application of the third rule is not possible as the transformation of an `Element e` requires that its predecessor has already been transformed before. That means that `Element e3` can only be transformed when `Element e2` already has been transformed. Similarly, `Element e2` can only be transformed when `Element e1` already has been transformed. Due to the NACs `Element e1` cannot be transformed at all[4]. Therefore, transferring the NACs from the declarative rules to the operational rules does not result in an applicable set of rules.

Dropping the NACs (and disregarding the given priorities for now) the application of the first rule of Figure 7.11c succeeds. We are now allowed to deal with each of the `Elements` of the `List` by applying the second rule of Figure 7.11c. The application of this rule to each of the `Elements` succeeds. Unfortunately, we end up with a target model that contains a `List` which in turn contains three `Elements` that are not related with each other which violates the constraints of our metamodel for `Lists`. That means that our transformation has lost the information on the successor relationships which is not intended. Therefore, dropping

[4]Even appointing that NACs only regard already transformed model elements does not improve the situation. Doing so would allow us to apply the second rule to all `Elements e1`, `e2`, and `e3`. This results in an invalid target model as explained below.

the NACs of the declarative rules when deriving operational rules also does not result in a proper set of rules.

When we regard the *priorities* of the rules shown in Figure 7.11c the transformation happens as follows. Firstly, we deal with the `List` element by applying the first rule as usual. For each `Element` of the list we have now to consider applying the third rule since it has a higher priority than the second rule. When we try to apply the third rule to `Element e3` this element has to be postponed until `Element e2` has been transformed. Correspondingly, the transformation of `e2` has to be postponed until `Element e1` has been transformed. The application of the third rule to `e1` is impossible as `e1` has no predecessor. Now we are allowed to apply the second rule to `e1` which succeeds. After that we can transform the postponed `Elements e2` and `e3` by applying the third rule twice. The resulting `List` now contains three `Elements` that are related to each other with the correct successor relationships. Thus, priorities allow us to correctly deal with a number of situations (if not all) where NACs would be desirable otherwise.

8 Realization

In this chapter we present the implementation of our approach as part of the MOFLON tool set. We start by giving a brief summary of MOFLON's history and its goals. Then we describe MOFLON's TGG support in detail. To this end we present the schema and the rule editors and how MOFLON generates code from a given TGG specification. Finally, we explain how the generated code can be applied in order to realize the desired integration of models.

8.1 The MOFLON meta-CASE tool

Driven by the needs of our industrial partners our Real-Time Systems Lab envisioned a framework for model-driven architecture (MDA) in 2003. We started by evaluating a number of different frameworks (c.f. Fig. 8.1) on top of which we planned to realize our vision. In the end we decided to realize our framework on top of the FUJABA tool-suite [Zün01]. First of all, FUJABA is open source and we already had contact to the main development groups at the universities of Paderborn and Kassel. Furthermore, FUJABA is written in Java and provides a plug-in mechanism in order to integrate additional extensions. Finally, a member of our group already had experience in developing such extensions for the FUJABA tool-suite. Therefore, at the 1st International FUJABA Days 2003 at the University of Kassel, Germany we presented our vision [AKRS03] to the FUJABA community.

Up to now FUJABA itself only provides a model editor based on a UML 1.x like metamodel. On the one hand we wanted to have a MOF 2.0-compliant editor instead. On the other hand we wanted to reuse FUJABA's Story Driven Modeling (SDM) editor which enables the user to graphically specify operation bodies based on the theoretical foundation of graph rewriting (c.f. Chapter 4). In order to achieve the latter we had to introduce an abstract interface layer in FUJABA on which SDM relies on and which is realized by FUJABA's original UML 1.x like editor as well as by our envisioned MOF 2.0-compliant editor. At the 2nd International FUJABA Days 2004 [Röt04] at Darmstadt, University of Technology,

	FUJABA	SfP	Together	Artisan	ECLIPSE	Rational Rose	Rational Rose/RT	ArgoUML/Poseidon	Dresden OCL-Compiler
Architecture Description Language	-	o	o	o	-	o	(+)	o	-
Integration Framework	-	o	-	(o)	(+)	(o)	-	-	-
Extensible Code Generator	o	+	o	o	-	-	-	-	-
Model Driven Architecture	+	(+)	-	(+)	-	-	+	-	-
Meta Modelling	o	o	(o)	-	(o)	-	-	(o)	-
OCL-Compiler	-	-	-	-	-	-	-	+	+
Rule Interpreter	+	-	-	-	-	-	-	-	-
Available Source Code	+	-	-	-	+	-	-	+/-	+
License costs	+	-	-	(-)	+	(-)	(-)	+/-	+

Figure 8.1: Tools and features for metamodeling taken from [AKRS03]

Germany, we presented the adaption of MOF 2.0 to this interface layer. In fact this adaption layer constantly is hard to maintain and still is a source for intricate bugs. Currently, the FUJABA community discusses possibilities to replace the different model editors by one UML 2.0-compliant editor and to get rid of the cumbersome adaption layer.

Moreover, FUJABA's code generation facility only generated Java code which complied to FUJABA's proprietary interfaces based on the Velocity template engine [1]. Rather, we want to generate Java code that complies to Sun's Java Metadata Interfaces (JMI) [Sun02] which describe a mapping from MOF 1.4 models to Java. First of all we analyzed if and how we could map the new MOF 2.0 features to JMI and found out that this could easily be done [ABS04]. After that we had to come up with an own set of Velocity templates in order to tweak FUJABA's code generation. In the meantime FUJABA provides a mechanism that allows to manage and easily exchange different sets of Velocity templates (e.g. Original FUJABA, JMI, EMF) at runtime. In fact MOFLON utilizes FUJABA's template mechanism to generate code from SDM operation specifications only. In order to generate JMI-compliant interfaces (and their implementations) from a MOF

[1] http://velocity.apache.org/

2.0-compliant model MOFLON relies on a component called MOMoC [Bic04] which already had been implemented independently from MOFLON beforehand. It is future work to replace one of both code generation facilities by the other in order to reduce maintenance overhead.

Finally, there already exists a plug-in for model-to-model integration based on TGGs for FUJABA [Wag01]. First of all this plug-in aims at the integration of two models that reside in FUJABA themselves. Rather, we aim at the integration of models that reside independently from each other in different CASE tools. Furthermore, the existing TGG plug-in specifies the metamodels of the to be integrated models as well as the metamodel of the traceability links in the same project. In the past that was the only possibility because FUJABA provided single project support only. The latest version of FUJUBA now provides basic multi-project support. We want to separate the metamodels of the to be integrated models from each other and from the metamodel of the traceability links in order to allow for the separate evolution of all involved metamodels. Moreover, the existing TGG plug-in relies on FUJABA's UML 1.x like metamodel for the specification of traceability link types. Concepts like *priorities* can only be specified using *stereotypes*. More sophisticated concepts like QVT's *where* dependency cannot be expressed at all. Finally, TGG rules in the existing TGG plug-in exist independently from elements (i.e., traceability link types) of the integration metamodel. As we have motivated in Chapter 6 we want to attach TGG rules to traceability link types. Therefore, we started to develop an entirely new TGG plug-in for MOFLON. It is future work to merge both TGG plug-ins with each other in order to provide a uniform support for TGGs in FUJABA / MOFLON.

Fig. 8.2 shows an overview of the architecture of MOFLON. The users can specify domain specific metamodels and tool representations with MOFLON's editors or import them from their favorite modeling tool (e.g. Rational Rose) via MOFLON's XMI import interface. Besides the metamodels which can be developed with the visual MOF 2.0 editor the users can also graphically specify operation bodies using the visual SDM editor which is based on graph transformations. Moreover, the TGG plug-in allows for the visual creation of declarative model integration specifications. From a MOF 2.0-compliant specification MOFLON can generate JMI-compliant interfaces as well as an in-memory implementation using MOMoC as stated above. In the upcoming version 1.2 of MOFLON the user can add OCL constraints to a metamodel from which MOFLON generates exe-

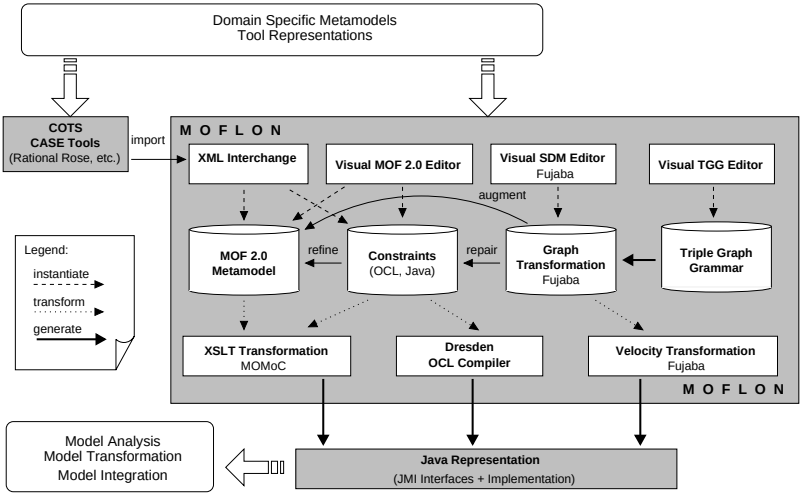

Figure 8.2: Overview of MOFLON's architecture

cutable Java code using the Dresden OCL compiler toolkit[2]. From SDM diagrams MOFLON generates JMI-compliant implementations for Java methods relying on the existing Velocity mechanism using an adopted set of Velocity templates. Finally, MOFLON derives a MOF 2.0-compliant specification from a declarative TGG specification in order to generate code. Thereby, TGG rules are translated into operational SDM diagrams. The resulting code can then plugged into exiting frameworks or can used to build new tools for model analysis, transformation, and integration. In the following we focus on the realization of model integration.

8.2 MOFLON TGG plug-in

Figure 8.3 depicts the architecture of our TGG plug-in for MOFLON. In order to specify the integration of two models the user initially appoints the metamodels of the regarded models. Conceptually, both metamodels may coincide (i.e.,

[2] http://dresden-ocl.sourceforge.net/

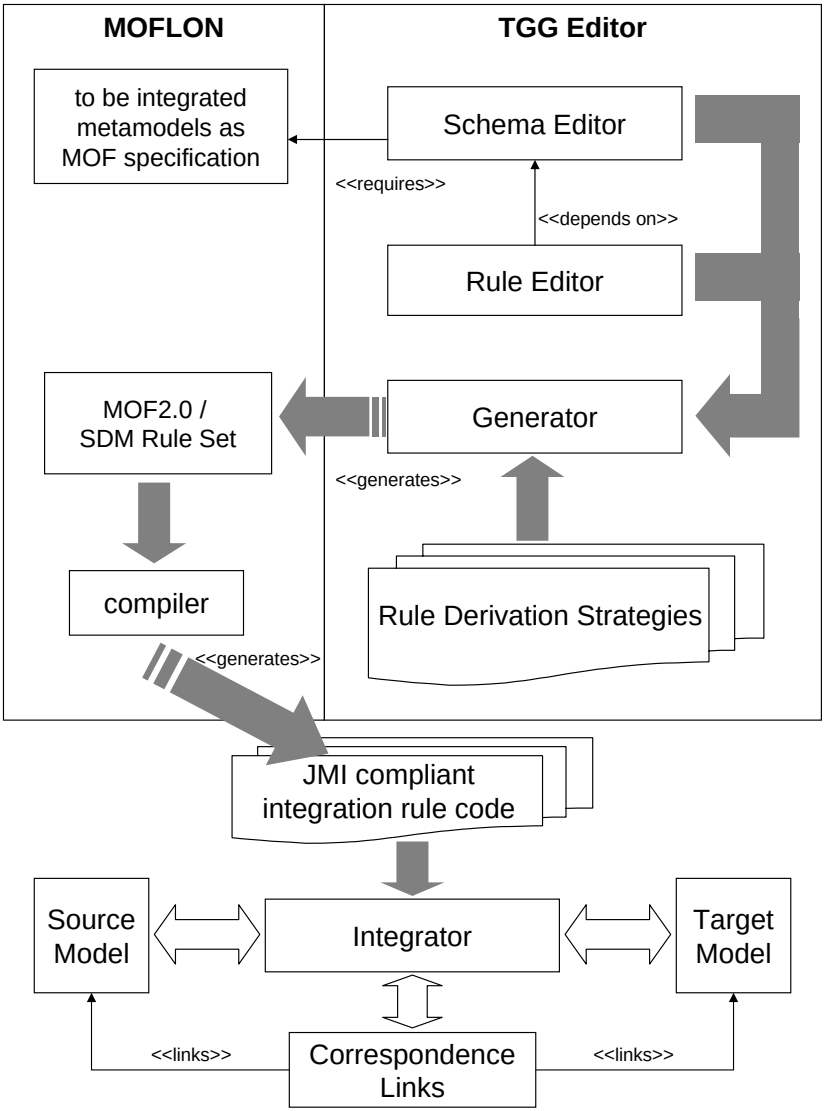

Figure 8.3: Architecture of MOFLON's TGG plug-in

both models conform to the same metamodel). However, this case is not covered by our implementation, yet. The metamodels reside as MOF 2.0-compliant specifications in MOFLON. The metamodels can be specified using MOFLON's MOF 2.0 editor or can be imported from a third party tool like Rational Rose using MOFLON's import facility. Using the schema editor of the TGG plug-in the user basically declares the integration link types that relate elements from both metamodels with each other. Furthermore, the users can structure and modularize their specification utilizing the provided package support. The user can specify a declarative TGG rule for each integration link type in order to express the desired model integration using the rule editor. For code generation purposes the users can translate their declarative TGG specification into a plain MOF 2.0-compliant specification. Thereby, the TGG schema is translated into a plain MOF 2.0 metamodel. Furthermore, each TGG rule is translated into a set of operational SDM rules which address the different model integration tasks. Finally, MOFLON's code generation facility is able to generate JMI-compliant Java code. This Java code can then be utilized by our integrator in order to integrate two given models with each other. Thereby, the integrator maintains a set of traceability links between elements of both models.

In order to implement our TGG plug-in we had two possibilities. First of all we had the choice to implement the plug-in based on the MOF metamodel as described in Chapter 5 and Chapter 6. To this end we would have written our plug-in as an extension to MOFLON's already existing MOF editor plug-in which implements the MOF metamodel. As we have already mentioned above FUJABA itself is not based on the MOF metamodel. Rather, FUJABA is based on a proprietary metamodel which is influenced by UML 1.5. Therefore, we would have had to implement an adaption layer from the MOF-based metamodel of our TGG plug-in to FUJABA's proprietary metamodel as we have done it for the MOF editor. Since we have learned that this adaption layer is hard to maintain we decided to postpone this option until FUJABA itself might be based on the MOF metamodel.

The other possibility was that we implement the TGG plug-in directly based on FUJABA's proprietary metamodel. On the one hand the price we have to pay for this option is that our implementation does only virtually meet our concept. On the other hand this option makes sure that our implementation integrates more naturally with FUJABA and is easier to maintain. Therefore, we decided to start with this option.

The TGG plug-in provides two editors. One editor allows for the specification of the schema of a TGG, i.e. the declaration of the desired correspondence link

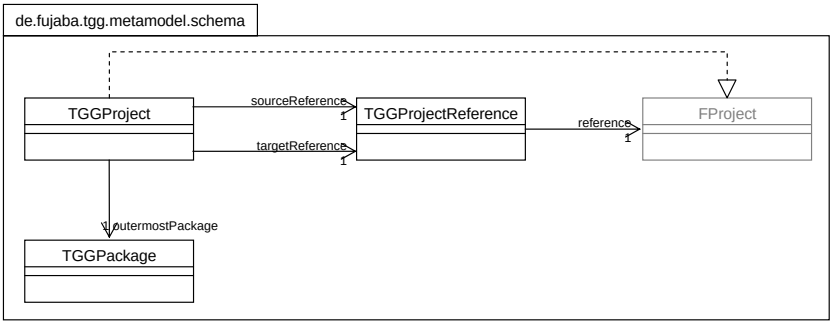

Figure 8.4: Project diagram

types. The other editor allows for the specification of declarative TGG rules. We emphasize that in our approach we attach TGG rules to correspondence link types. Thereby, each correspondence link type can have at most one TGG rule. In the following we explain the metamodel of our TGG plug-in.

Since the actual metamodel is far too complex we do the following simplifications. As we have chosen to base our TGG plug-in directly on FUJABA's metamodel our metamodel relies on FUJABA's so-called *FInterfaces*. The FInterfaces constitute FUJABA's metamodel and have been introduced to reflect the similarities of FUJABA's former UML 1.5-like metamodel and our intended MOF 2.0 metamodel. In the figures we only show those FInterfaces that are necessary. Furthermore, FUJABA provides some abstract implementations for some of its FInterfaces. To keep things simple we entirely disregard these abstract implementations. Finally, the actual implementation contains a number of associations that are needed only for technical reasons. In our description we abstract from them and show only those associations that are relevant.

8.2.1 The TGG schema editor

FUJABA's means to manage single specifications are projects. Therefore, our TGG plug-in introduces a new type of project, i.e. TGG projects as shown in Figure 8.4. Each `TGGProject` refers to two `FProjects` by means of two `TGGProjectReferences`. As `FProject` is FUJABA's most abstract spec-

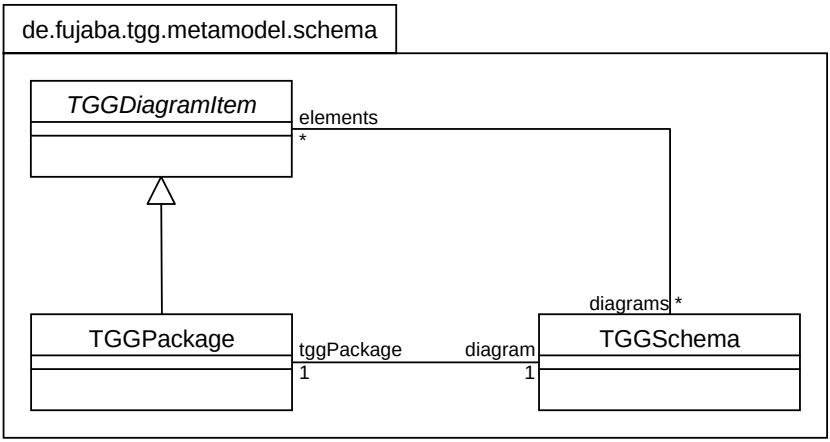

Figure 8.5: Schema diagram

ification of a project our plug-in basically is able to refer to any project in FUJABA. The referred `FProjects` contain the metamodels of the to be integrated models. Currently, there are three types of projects provided by FUJABA. First of all, `UMLProjects` as introduced by FUJABA itself are used to specify UML 1.5-like models. Secondly, `MOFProjects` as introduced by MOFLON's MOF editor are used to specify MOF 2.0-compliant models. Finally, the just introduced `TGGProjects` are used to declaratively specify QVT-like model integrations based on TGGs. Although it is conceptually possible and meaningful to integrate TGG specifications with other projects our plug-in explicitly disallows a `TGGProject` to refer to other `TGGProjects` in order to keep things simple for the user. We can easily remove this restriction in the future. Furthermore, each `TGGProject` is provided with a distinguished `TGGPackage` which represents the outermost package containing the whole TGG specification.

In order to visualize models FUJABA declares diagrams and diagram items that are contained in diagrams. In our TGG plug-in each `TGGPackage` is provided with a corresponding diagram called `TGGSchema`[3] which displays the content of the `TGGPackage`. Each `TGGSchema` contains an arbitrary number of

[3] In upcoming releases of our TGG plug-in `TGGSchema` should be renamed to `TGGDiagram`.

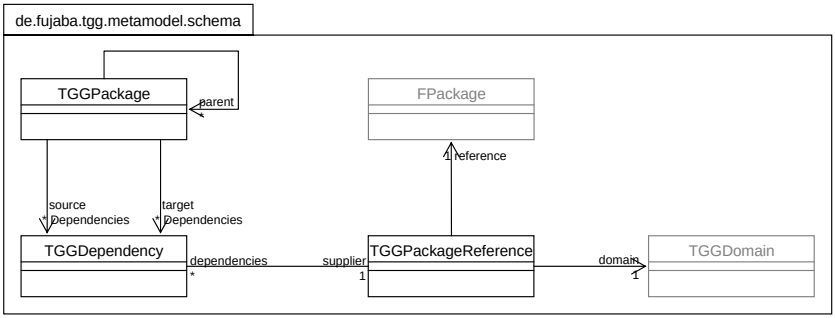

Figure 8.6: Package diagram

TGGDiagramItems. The other way around each *TGGDiagramItem* can be contained in an arbitrary number of TGGSchema. Thereby, *TGGDiagramItem* is an abstract class that represents any item that has a visual representation in our plug-in. For instance TGGPackages themselves are *TGGDiagramItems*. Actually, the majority of classes declared by the metamodel of the TGG plug-in are *TGGDiagramItems*.

As we have seen in Chapter 5 packages are used to modularize specifications into possibly reusable parts. In the initial version of our TGG plug-in TGGPackages can be nested as depicted in Figure 8.6. Furthermore, TGG-Packages can refer to FPackages by means of TGGPackageReferences. Conceptually, TGGPackageReferences should behave like *import* dependencies. As such the elements of a TGGPackage can see the elements of each referred, i.e. imported, FPackage. Actually, in the current implementation these references are not evaluated and are only used for documentation. Rather, all elements of the integragted FProject are globally considered visible. Upcoming versions of the TGG plug-in should respect these dependencies accordingly. TGGPackageReferences are connected to the importing TGGPackages by TGGDependencies. Finally, each TGGPackageReference is provided with a TGGDomain which states whether the imported package belongs to the source or the target project.

Figure 8.7 introduces the central class of a TGG schema called TGGNode. Each TGGNode declares a type of correspondence links. Conceptually, each

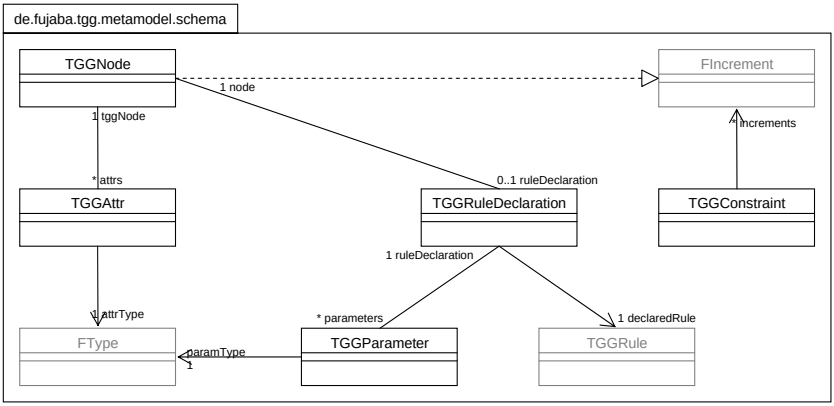

Figure 8.7: Node diagram

`TGGNode` inherits its properties from class and association. As a class each `TGGNode` carries an arbitrary number of attributes called `TGGAttrs`. Each attribute has a type (`FType`). The attributes can be used to store meta-information on correspondence links as *date of creation*, *name of creator*, and so on. Furthermore, these attributes can store technical information on the model integration process. Additionally, each `TGGNode` can be provided with up to one `TGGRuleDeclaration` that declares a `TGGRule`. `TGGNodes` can be marked as *abstract*. `TGGNodes` that do not provide a `TGGRuleDeclaration` inherently must be marked as *abstract*. Abstract `TGGNodes` cannot be instantiated directly. Rather, abstract `TGGNodes` must be specialized as explained below. Each `TGGRuleDeclaration` has an arbitrary number of typed `TGGParameters` that are used in the corresponding `TGGRule` as explained in Chapter 6. Moreover, `TGGRuleDeclarations` define the *priority* of the attached `TGGRule` which is considered at rule application time as described in Section 7.2. Since `TGGNode` implements the `FIncrement` interface `TGGNodes` can be attached with `TGGConstraints`. `TGGConstraints` constitute (OCL) constraints that express further conditions that must hold in order to consider a `TGGNode` as consistent.

As illustrated in Figure 8.8 `TGGNodes` can inherit from each other by means of `TGGGeneralizations`. Currently, our implementation does not enforce

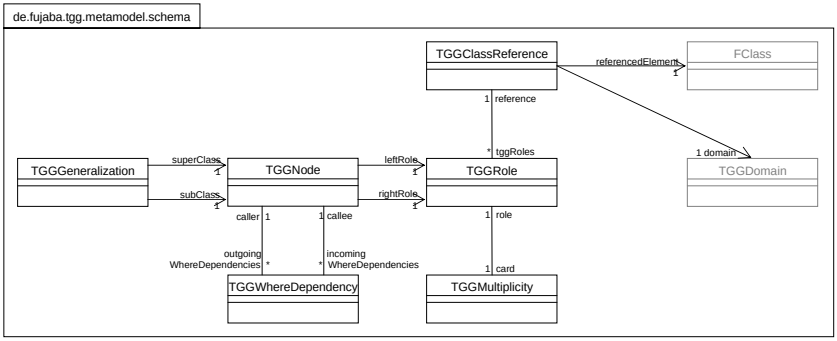

Figure 8.8: Node dependencies diagram

the conceptual restrictions which generalization relationships imply for the underlying TGG rules as pointed out in [KKS07]. Furthermore, `TGGNodes` can be related to each other by `TGGWhereDependencies`. A `TGGWhereDependency` expresses that the attached rule of the calling `TGGNode` invokes the application of the rule of the called `TGGNode`. Since `TGGNodes` inherits properties from associations each `TGGNode` has a *left* and a *right* `TGGRole`. Each `TGGRole` is provided with a `TGGMultiplicity`. `TGGMultiplicities` express how many elements of one model are linked to one element of the other model and vice versa. Furthermore, a `TGGRole` refers to a `FClass` by means of a `TGGClassReference`. The `TGGClassReference` refers to the type of model element that is linked by the `TGGNode`. Additionally, a `TGGClassReference` is provided with a `TGGDomain` which states whether the linked model element belongs to the source or the target model.

Figure 8.9 depicts a screenshot of our TGG schema editor. The project tree is shown on the left-hand side. The tree contains all projects that currently have been opened with FUJABA/MOFLON and their respective contents. The editor pane is shown to the right-hand side. Currently, the editor pane depicts the schema of a showcase which will be discussed in Section 9.2 in detail.

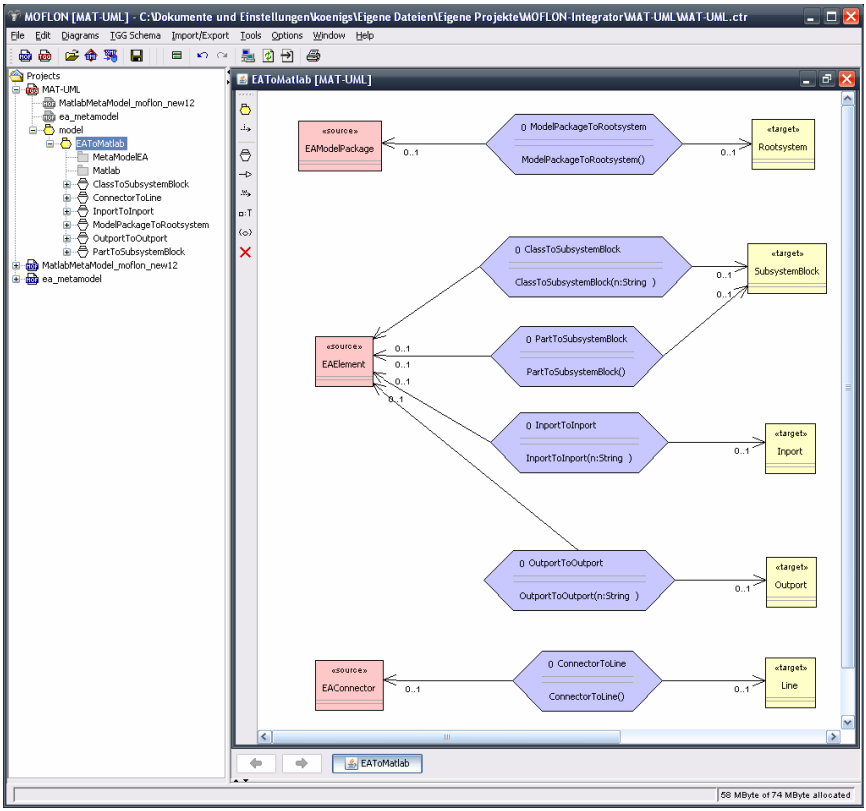

Figure 8.9: Screenshot of the TGG schema editor

8.2.2 The TGG rule editor

Besides the schema editor our TGG plug-in provides an editor for declarative TGG rules. Figure 8.10 depicts the metamodel of the TGG rule editor. Each `TGGRule` which is declared in the schema as explained above consists of a number of `FElements`, i.e. `TGGObjects`, `TGGLinks`, or `TGGIntegrationLinks`. A `TGGObject` is an object pattern that is to be matched either in the source or in the target model as specified by the attached `TGGDomain`. Cor-

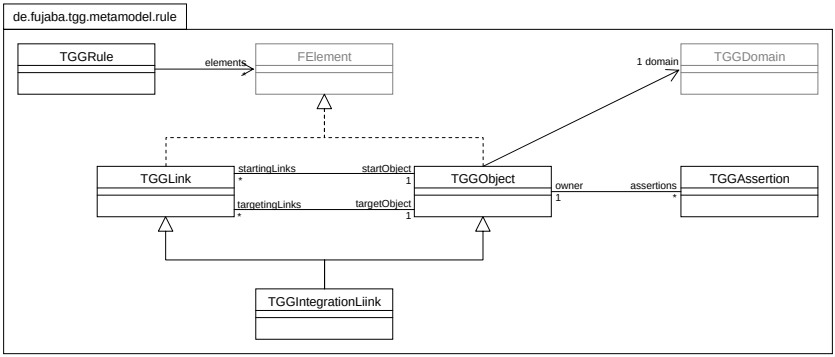

Figure 8.10: Rule diagram

respondingly, a `TGGLink` is a link between objects of either the source or the target model. `TGGIntegrationLinks` represent correspondence links that link objects of the source with objects of the target model. Both, `TGGObjects` and `TGGIntegrationLinks` can be attached with `TGGAssertions`. A `TGGAssertion` is either an attribute condition or an attribute assignment that virtually is to be evaluated at rule application time.

Figure 8.11 depicts a screenshot of our TGG rule editor. This time the editor pane shows a declarative TGG rule of a showcase which will be discussed in Section 9.2 in detail.

8.2.3 Code generation

In order to generate executable code from a declarative TGG specification we perform the following two steps. First of all, we translate the declarative TGG specification into a plain MOF 2.0-compliant specification. Secondly, we generate JMI-compliant Java code from the resulting MOF project using MOFLON's common code generation facility as it is. Therefore, the MOF project can be considered as intermediate code. From the schema of the TGG specification we derive a MOF 2.0-compliant metamodel. From the declarative TGG rules we derive operational SDM graph rewriting rules.

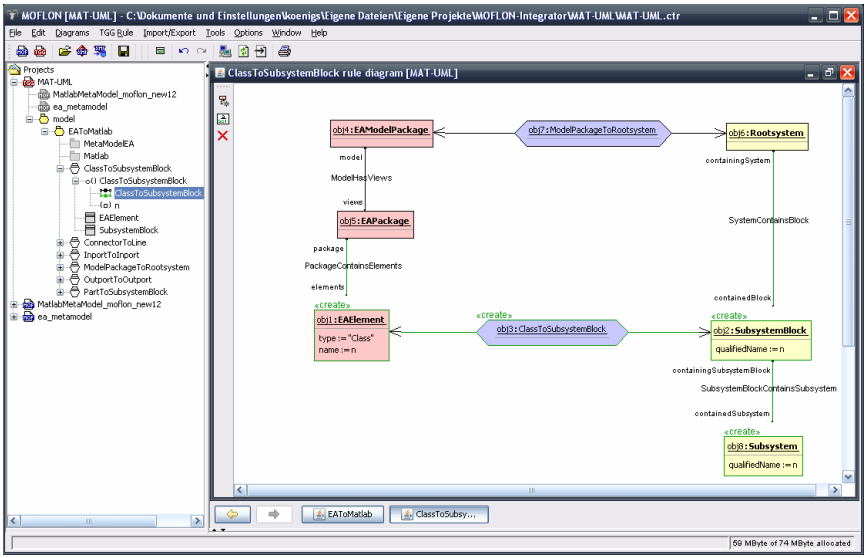

Figure 8.11: Screenshot of the TGG rule editor

Particularly, the translation of a TGG specification into a plain MOF 2.0-compliant specification looks as follows. First of all, we translate the hierarchy of `TGGPackages` into a corresponding hierarchy of `MOFPackages`. Thereby, we translate dependencies between `TGGPackages` (i.e., *imports*) accordingly. Each `TGGNode` is translated into a `MOFClass` and two `MOFAssociations`. The `TGGAttrs` of the `TGGNode` are translated into `MOFAttributes` of the corresponding `MOFClass`. `TGGConstraints` that are attached to a `TGGNode` are translated into `MOFConstraints` that are attached to the corresponding `MOFClass`. `TGGRuleDeclarations` and `TGGParameters` are not directly translated. Rather, the `TGGRule` attached to a `TGGRuleDeclaration` is translated into a number of operational SDM rules as explained below. Thereby, `TGGParameters` are resolved as possible. Furthermore, `TGGGeneralizations` between two `TGGNodes` are translated into `MOFGeneralizations` between the corresponding `MOFClasses`. `TGGWhereDependencies` are not directly translated at metamodel level. Rather, `TGGWhereDependencies` are regarded at operational rule derivation time as explained below. `TGGClassRef-`

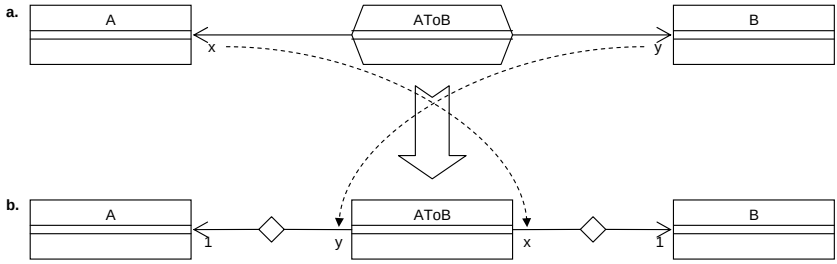

Figure 8.12: Translation of TGGMultiplicities

erences are translated into MOFClasses that are provided with MOFTags which instruct MOFLON's code generator to generate the desired references to classes of the integrated external metamodels. TGGRoles and TGGMultiplicities are considered during the creation of the two MOFAssociations that correspond to a TGGNode. The TGGRoles determine to which MOFClasses the MOFClass that corresponds to the regarded TGGNode is associated with.

TGGMultiplicities are dealt with as illustrated in Figure 8.12. The multiplicities in Figure 8.12a. mean that each instance of class A is linked with y instances of class B by correspondence links of type AToB. The other way around each instance of class B is linked with x instances of class A. According to Figure 8.12b. AToB is translated into a class AToB and two associations. The multiplicities in Figure 8.12b. mean that an instance of class A is linked to y instances of class AToB. Furthermore, each instance of class AToB is linked with one instance to class B. Therefore, each instance of class A still is indirectly linked to y instances of class B. The other way around each instance of class B is linked with x instances of class AToB. Each instance of class AToB is linked with one instance of class A. Therefore, each instance of class B still is indirectly linked to x instances of class A.

Besides the TGG schema the translation into a MOF 2.0-compliant specification deals with the declarative TGG rules. Each declarative TGG rule is translated into a set of operational SDM rules as conceptually explained in Section 7.1. Figure 8.13 presents the forward transformation SDM rule which has been derived from the declarative TGG rule from Figure 8.11. Our implementation adds a set

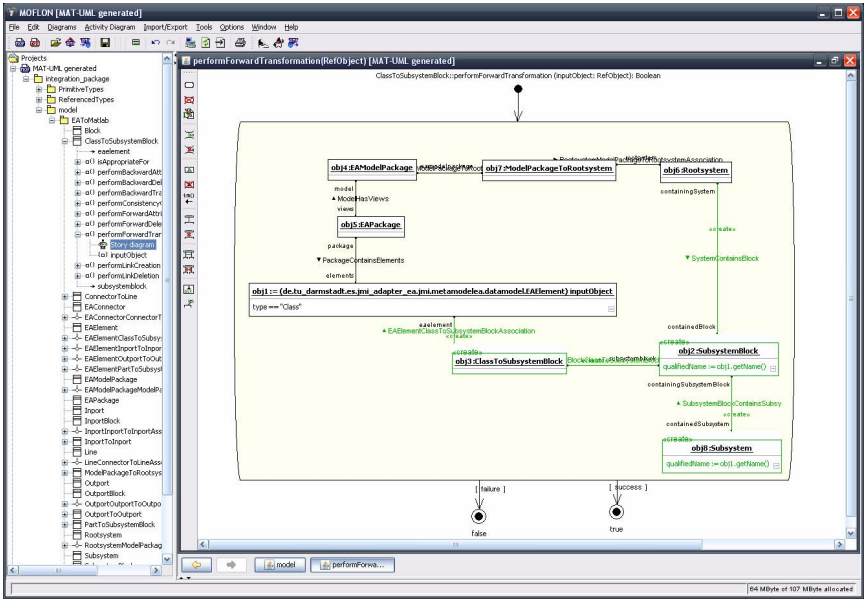

Figure 8.13: Screenshot of the SDM rule editor

of `MOFOperations` for each `TGGRule` to the `MOFClass` that corresponds to the `TGGNode` that owns the rule. Furthermore, each `MOFOperation` is provided with a SDM diagram that represents the operational rule. Finally, from the generated MOF 2.0-compliant specification MOFLON's code generator is able to generate JMI-compliant Java code that can then be applied to realize the intended model integration.

8.3 Integrator

Besides a declarative TGG specification from which we generate executable Java code we need a framework which applies (i.e., executes) the generated code in order to perform the specified model integration. To this end the framework must be

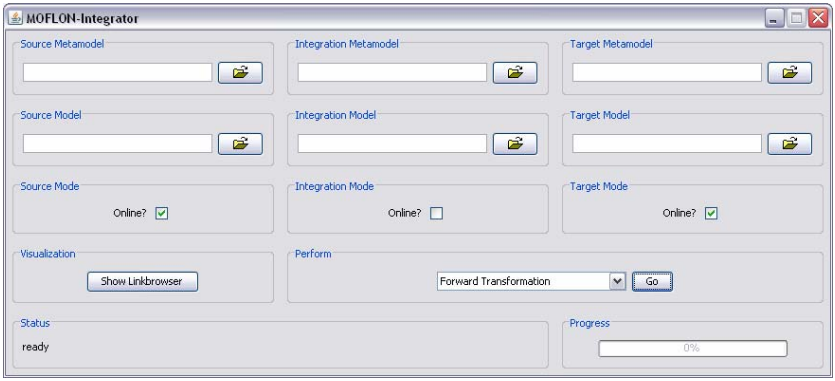

Figure 8.14: Screenshot of the MOFLON-Integrator

able to access the generated code. Furthermore, the framework must be provided with strategies how to apply the generated code (cf. Section 7.2).

Since the framework must be able to access the generated code we have two possibilities how to implement such a framework. On the one hand we could generate a dedicated framework for each model integration, which is inseparably linked to the code generated from the TGG specification. This is done by approaches like [Bec08] for instance. On the other hand we could implement a generic framework that is able to dynamically access the generated model integration code at runtime. Since both possibilities are appropriate and feasible it is merely a matter of taste. In the end we chose the latter possibility for implementing a prototypical model integration framework. The name of the prototype we implemented simply is *MOFLON-Integrator*. We emphasize that the MOFLON-Integrator is a component that runs independently from the MOFLON case tool.

Figure 8.14 presents a screenshot of our prototype. In the first row of the MOFLON-Integrator dialog we have to specify the metamodels of the source, the target, and the correspondence models. In the second row we have to specify the source, target, and correspondence models. In the third row we have to choose whether we want to access the regarded models through the APIs (application programming interfaces) of the CASE tools that keep the data (*online mode*) or through *xmi*-exports of the tools' data (*offline mode*). In case of the

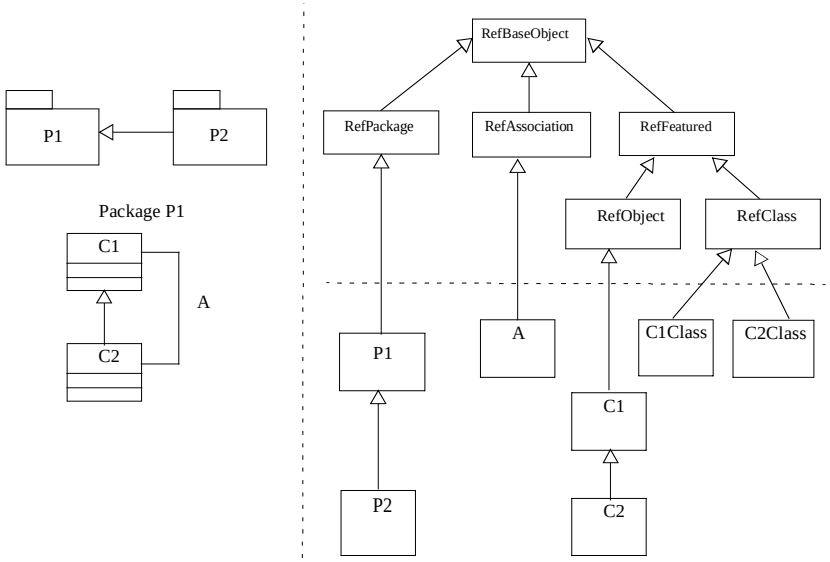

Figure 8.15: Mapping of a MOF model to Java interfaces (taken from [Sun02])

correspondence model the online mode could mean that the correspondence links are stored in a database instead of a simple *xmi*-file.

Generally, the offline mode requires *jar*-files that have been generated by MOFLON from MOF 2.0-compliant specifications. These *jar*-files contain JMI[4]-compliant interfaces [Sun02] and corresponding in-memory implementations. In case of the online mode these *jar*-files contain JMI-compliant interfaces as well. Rather than corresponding in-memory implementations these *jar*-files provide implementations that map the JMI interfaces to the APIs of the regarded tools. Both variants of *jar*-files (online or offline) usually provide a reader and a writer in order to (de-)serialize the tools' data using the *XMI*[5] file format [OMG05a].

Sun's JMI specification defines a mapping of MOF models to Java interfaces. Figure 8.15 exemplarily illustrates this mapping. On the left-hand side we see

[4] *Java Metadata Interfaces*
[5] XML metadata interchange

a very simple MOF model. We have two packages `P1` and `P2`. Thereby, `P2` inherits from `P1`. Furthermore, `P1` contains two classes `C1` and `C2` that inherit from each other and are related to each other by means of an association `A`. The right-hand side of the figure depicts the corresponding JMI interfaces. First of all the JMI specification defines a set of reflective interfaces that are independent from the actual metamodel. These interfaces enables users to explore the corresponding metamodel as well as conforming models without any knowledge of the metamodel itself. Furthermore, the JMI specification demands that for a given metamodel a number of dedicated interfaces must exist that enable the users to access the metamodel and conforming models relying on the actual types that are defined by the metamodel.

All other JMI interfaces directly or indirectly inherit from the reflective interface `RefBaseObject`. This interface contains operations for querying the identity of model elements, to determine their meta object and their containing package, and to evaluate constraints attached to them. The `RefPackage` interface provides operations for accessing elements (i.e., classes, associations, nested packages that reside in a given package. The operations of the `RefAssociation` interface enables its users to query and maintain links between model elements that are based on a given association. Using the operations of the `RefFeatured` interface a user can utilize the (possibly statical) features (i.e., attributes and operations) of model elements. Additionally, the `RefObject` interface provides operations to check whether a model element is of a certain type and to determine the containment hierarchy of model elements by means of composition relationships. Finally, the `RefClass` interface offers operations to query the set of model elements of a given type and allows for the creation of new model elements.

Besides the reflective interfaces a number of metamodel-dependent interfaces that extends the reflective interfaces are generated. For each package of the metamodel a corresponding interface `P1` and `P2` are generated. Similarly, association `A` is mapped to a corresponding interface. Finally, for each class `C1` and `C2` two interfaces are generated. The interfaces `C1` and `C2` represent the actual model elements whereas the interfaces `C1Class` and `C2Class` act as factory interfaces that additionally keep track on the set of model elements of the regarded type. These factory interfaces are called *proxies* in the JMI specification.

Our generic model integration framework heavily relies on the reflective JMI interfaces since the framework should be kept independent from the actual metamodels of the to be integrated models. In contrast the code generated from the

declarative TGG rules relies on the typed JMI interfaces as the rules are only valid for the metamodels chosen at specification time anyway.

The fourth row of Figure 8.14 contains a button for invoking the *Linkbrowser* of the MOFLON-Integrator as explained in the following section. Furthermore, there is a combo box that allows the users to choose the model integration task they want to perform. Currently, only the forward model transformation task is prototypically implemented. The implementation of the remaining integration tasks is straight-forward and will be include in the next version of the MOFLON-Integrator. Finally, the *Go*-button invokes the chosen model integration task which results in the application of the appropriate model integration rules according to the rule application strategy as explained in Section 7.2. The last row of the MOFLON-Integrator dialog is used to display information on the currently running integration task.

8.4 Linkbrowser

Besides the fact that correspondence links are maintained in order to realize model integration the users want to be able to inspect existing correspondences for traceability purposes. In fact the visualization of correspondence links is one of the main purposes of the Toolnet framework as mentioned in Section 9.1. Since in common software and system development projects the number of involved models, model elements, and correspondences is large the question arises how to appropriately display their relationships. In [FK03] we identified the following use cases that should be addressed by a visualization component.

1. The user wants to get a complete overview of a considered project. In particular, this is important for project managers which have to coordinate development teams, monitor the project's progress, and plan the next steps.
2. A pair of developers is working on a certain aspect of a system and want to discuss problems at a very fine-grained level of abstraction. Therefore, they need to inspect the correspondences at the level of model elements.
3. A user modifies a part of a considered system and wants to predict the impact of his modifications to the rest of the whole system.
4. Users want to maintain correspondence links using one component that allows to access all links of the considered project.

The solution we presented in [FK03] was based on *XML Topic Maps*[6] and relied on the *Touchgraph*[7] framework. Topic Maps as defined by the *ISO/IEC 13250 Topic Maps* standard[8] are designed for describing large knowledge structures. The basic concepts are typed *Topics* and typed *Associations* that relate topics with each other. The mapping of model elements to topics and correspondence links to associations is obvious and straight-forward. We used the Touchgraph framework to visualize Topic Maps. The Touchgraph framework displayed the model elements and correspondence links in a so-called hyperbolic manner. This means that a limited number of model elements and their correspondences are in the focus of the visualization. These elements are displayed in detail. The remaining elements are displayed in a more abstract and vague way at the border of the visualization area. The user is able to shift the focus by navigating along the correspondence links. Thus, the proposed solution addressed the use cases 1, 2, and 3. In [FK03] we did not address Use Case 4. Although the use of a hyperbolic view seemed to be a good idea at first glance practice has shown that in fact this kind of view is not very convenient as the user can explore the direct neighborhood of a model element and its attached traceability links only. The user is not provided with an overview of all existing traceability links.

Therefore, we propose another visualization component based on Fraunhofer's *Matrixbrowser*[9]. Basically, the Matrixbrowser is used to visualize a number of elements called *nodes* and links between them called *relations*. The elements are stored in a tree structure as illustrated on the left-hand side of Figure 8.16. The links are visualized in a matrix as shown on the right-hand side of Figure 8.16. To this end one subtree is displayed vertically, a second is displayed horizontally. Links are depicted as circles in the matrix that are connected to the linked elements by arrows. Using drag and drop the user is enabled to arbitrarily choose subtrees he wants to inspect. Particularly, the user may choose to select the entire tree for both vertical and horizontal display. Although the resulting matrix is large it contains all links that exist between elements of the tree. Obviously, we map model elements to nodes while correspondence links are mapped to relations. In order to construct the needed tree structure we create a root node which represents the considered project as a whole. Attached to this root node we have nodes that represent all relevant data sources of the project. Each data source node is

[6] http://www.topicmaps.org/xtm/
[7] http://www.touchgraph.com
[8] http://topicmaps.org/xtm/index.html
[9] http://matrixbrowser.hci.iao.fraunhofer.de/en/

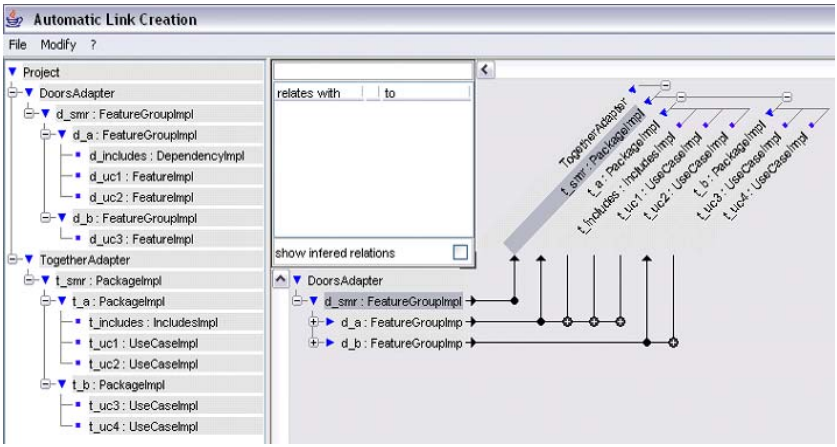

Figure 8.16: Screenshot of the Matrixbrowser

then attached with the hierarchy (by means of composition relationships) of its contained model elements. Now the user is enabled to select arbitrary parts of a project (e.g. the whole project, single data sources, parts of data sources) in order to inspect existing correspondence links.

For enhancing the expressive power of the visualization we added the following modifications to the Matrixbrowser. The resulting component is called *Linkbrowser*.

1. The background color of nodes can be changed. For instance this allows to display consistency information for each model element (e.g., green signals a consistent model element, red signals an inconsistent model element, yellow means that it is unknown whether a model element is consistent).
2. The color of relations can be changed. Correspondingly, this allows to display consistency information for each correspondence links using the same color schema as above.
3. Nodes and relations can be attached with context menus. This allows for invoking model integration rules on single model elements and correspondence links. For instance the consistency of a correspondence link which consistency status is yet unknown can be (re-)calculated.

4. The tooltips of nodes and relations can be modified. This allows for displaying additional information on each model element and correspondence link.

5. It is possible to replace the circles that represent correspondence links in the matrix by arbitrary graphical elements. This is useful when crucial information on a correspondence link should be displayed in the matrix.

6. Finally, a problem panel as inspired by the problem panel of the Java Development Toolkit of the Eclipse framework has been added. In this panel further information a consistency violation is listed. When the user clicks on an entry of this list the concerned model element or correspondence link is selected from the matrix.

Currently, the features listed above are only included as part of the ToolNet framework. It is ongoing work to transfer these features to the Linkbrowser of the MOFLON-Integrator. The Linkbrowser has proved to be useful for cases where the user wants to inspect correspondence links of two parts of a project. It is an open issue to provide an appropriate support for navigating along existing correspondence links between various parts of a project (e.g., navigation from a requirement, to its corresponding specification, to its corresponding realization, to its corresponding test case).

9 Application

In this chapter we present the application of our approach to the use cases identified in Chapter 1.3. Particularly, we demonstrate the traceability link creation and model-model consistency analysis in the context of the ToolNet project [ADS02]. Furthermore, we present the transformation of an Enterprise Architect[1] model into a Matlab/Simulink[2] model. The entire set of TGG rules concerning our running example from Section 3.1 can be found in Appendix A.

9.1 The ToolNet project

ToolNet [ADS02] has been developed by DaimlerChrysler in cooperation with the universities of Berlin, Paderborn, and Darmstadt. Currently, ToolNet is distributed as a product by Extessy[3]. The aim of ToolNet is to provide a tool integration platform. Thereby, tool integration means relating model elements which are kept in an arbitrary number of commercial off the shelf (COTS) tools with each other by means of traceability links.

To this end ToolNet offers a Java-based framework for implementing tool adapters. Each ToolNet tool adapter conforms to the proprietary ToolNet tool adapter interface. On the one hand this interface defines a number of user interface and tool management related operations such as *highlight a certain object* which is used to focus on model elements that are linked with each other, *marked linked objects* which marks all model elements of a given tool that are linked with other model elements of different tools by traceability links, or *start* and *stop* a certain tool. On the other hand this interface defines basic read-only operations for accessing the model data stored in the adapted tool similar to but not as complete as the read-only operations defined by the JMI standard as introduced in the preceding chapter. Moreover, ToolNet provides a component called *ToolNet*

[1] http://www.sparxsystems.de/
[2] http://www.mathworks.com/
[3] http://www.extessy.com/

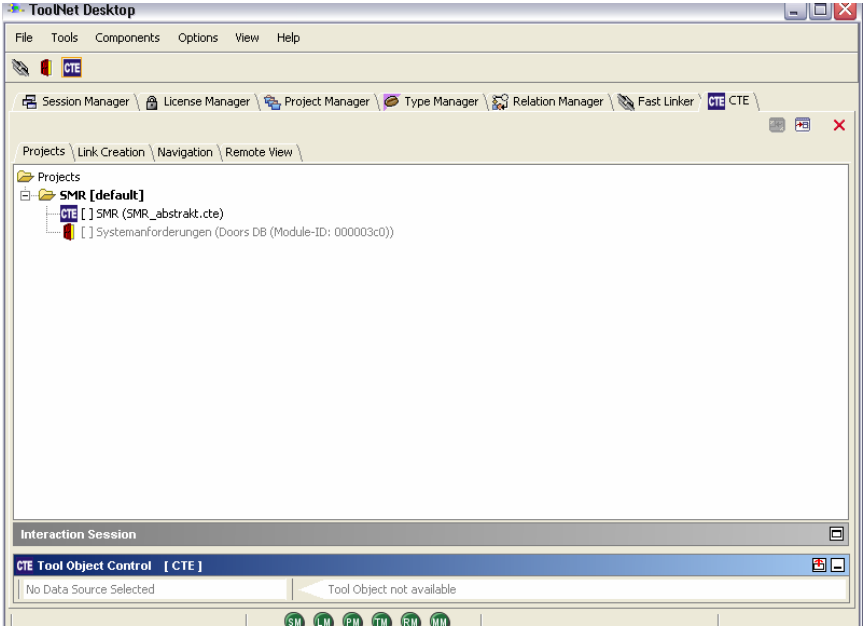

Figure 9.1: Screenshot of the ToolNet desktop

Desktop which is a central visual component that allows the user to invoke tool integration related operations such as *start* or *stop* a certain tool, *query* for existing traceability links, *calculate* the model elements that are linked to a given model element, and so on. Figure 9.1 shows a screenshot of the *ToolNet Desktop*. Finally, ToolNet has a database called *Relation Repository* for maintaining created traceability links. Note that ToolNet rather than other tool integration scenarios only stores traceability link information in a (central) database. The model elements created with the adapted tools remain in their tool repositories.

One of our contributions to ToolNet is a component called *Automatic Link Creation*. This component utilizes Java code that has been generated from operational *Correspondence link creation* rules as introduced in Chapter 7. To recall, these operational rules are derived from a corresponding declarative TGG rule as introduced in Chapter 6. The *Automatic Link Creation* component enables ToolNet

to automatically calculate correspondences of model elements that are stored in the adapted tools. Another contribution is the *Linkbrowser* component (cf. Section 8.4) that allows for the creation and visualization of and navigation on the traceability links stored in the *Relation Repository*.

Concerning the actual rules we specified for the ToolNet marketing showcase we experienced that the rules which DaimlerChrysler had in mind were very simplistic. The showcase deals with the integration of a *Doors*[4] requirements specification and a CTE[5] test case specification. The aim of the showcase is to link requirements from Doors with the test cases in CTE which are meant to test the regarded requirement. Thereby, DaimlerChrysler focused on attribute values only and disregarded the structure (e.g. the hierarchy information) of the involved models. Thus, the rules of the showcase are not adequate for illustrating the strengths of our TGG approach. Rather, the showcase is suitable for demonstrating the automatic application of the Java code which has been generated from the corresponding declarative TGG rules. However, it turned out that the users of ToolNet are skeptic concerning the automatic creation of traceability links. Although, the *Automatic Link Creation* engine performed well most users were content with creating and maintaining traceability links manually. The strengths of the automatic creation of traceability links become apparent when the number of to be created and maintained traceability links becomes too large for manual processing.

Figure 9.2 illustrates a specified rule of the ToolNet showcase. Figure 9.2a. depicts the declaration of the TGG link type `RequirementToTestcase`. The link typed connects `Requirements` of Doors with `Testcases` of CTE. Each `Requirement` of Doors should be linked with at least one `Testcase` of CTE. The other way around, each `Testcase` should be linked with exactly one `Requirement`. The constraint which is attached to the integration link type states that each existing integration link should be considered as inconsistent if the text of the `Requirement` does not contain the name of the `Testcase`. To recall this constraint is only evaluate at consistency checking time rather than at link creation time. Figure 9.2b. depicts the declarative TGG rule. The rule simultaneously creates a `Requirement` in Doors and a `Testcase` in CTE. The `id` of the `Requirement` as well as the `testing_id` of the `Testcase` are assigned to the provided input parameter `i`. Figure 9.2c. depicts the corresponding

[4] http://www.telelogic.de/products/doors/index.cfm
[5] http://www.systematic-testing.com/

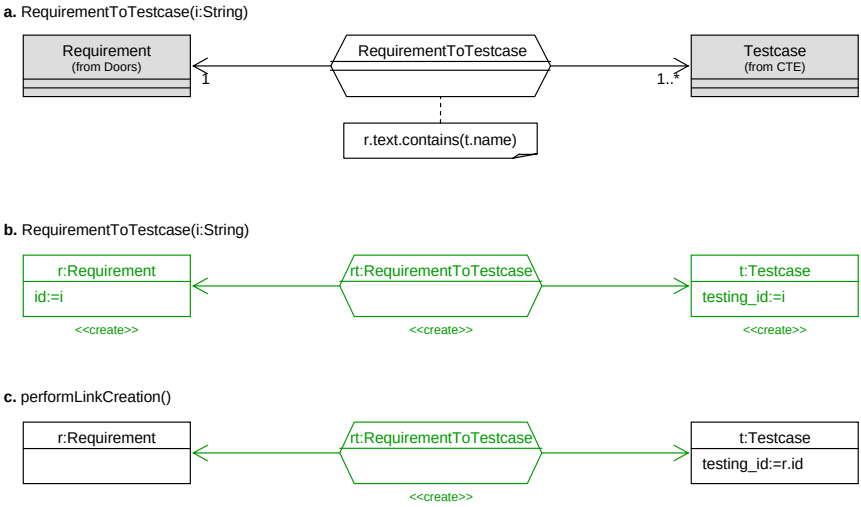

Figure 9.2: Example of an integration rule of the ToolNet showcase

derived operational correspondence link creation rule. This rule creates an integration link between a `Requirement` and a `Testcase` if the `testing_id` of the `Testcase` matches the `id` of the `Requirement`.

9.2 Enterprise Architect to Matlab/Simulink transformation

In order to demonstrate the strengths of our MOFLON meta-CASE tool in general and its model-model integration solution in particular we prepared an own showcase[6] [KAK+08] which is independent from ToolNet. This showcase deals with the transformation of an Enterprise Architect composite structure diagram of a simple cruise control system into a corresponding Matlab/Simulink model. Again, we access the models stored in each tool through tool adapters. This time we rely on JMI-compliant read/write tool adapters.

[6] Further showcases and information can be found here: http://moflon.org/documentation/tutorial.html

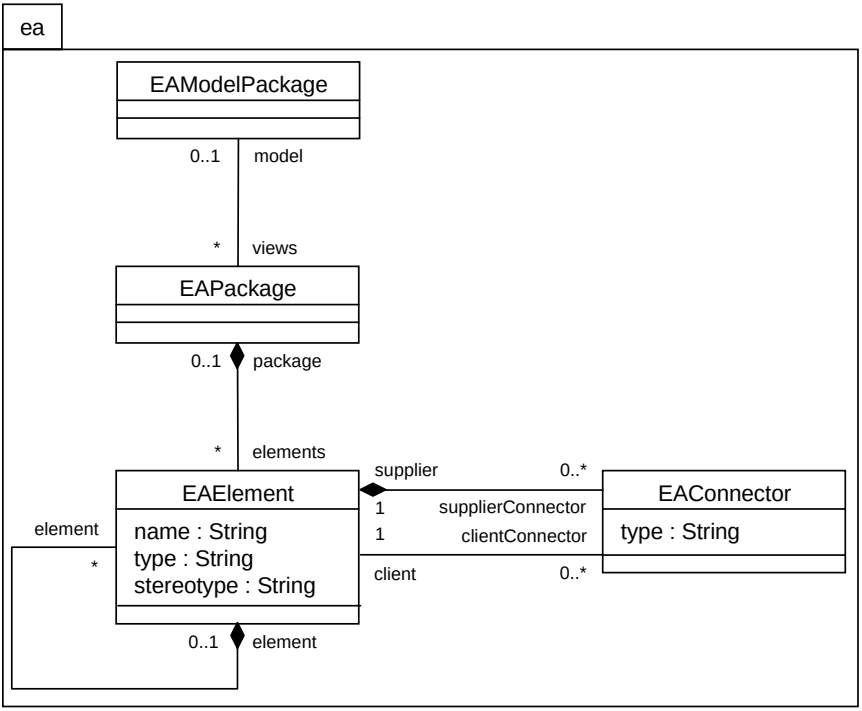

Figure 9.3: Simplified metamodel of Enterprise Architect

Figure 9.3 depicts a simplified metamodel of Enterprise Architect. Basically, an Enterprise Architect model is contained in an `EAModelPackage`. An `EAModelPackage` is provided with a number of `EAPackage` which represent `views` of the model. Among others each `EAPackage` contains a number of `EAElements` which have `names`, `types`, and `stereotypes`. Thereby, the values of `type` and `stereotype` determine which actual (UML) element is meant and how it is graphically represented. For instance an outport is represented as an `EAElement` which `type` is `Port` and which `stereotype` is `Outport`. Each `EAElement` may contain further `EAElements`. Finally, `EAElements` can be connected to each other by typed `EAConnectors`. The source of an `EAConnector` is called supplier; the target is called client.

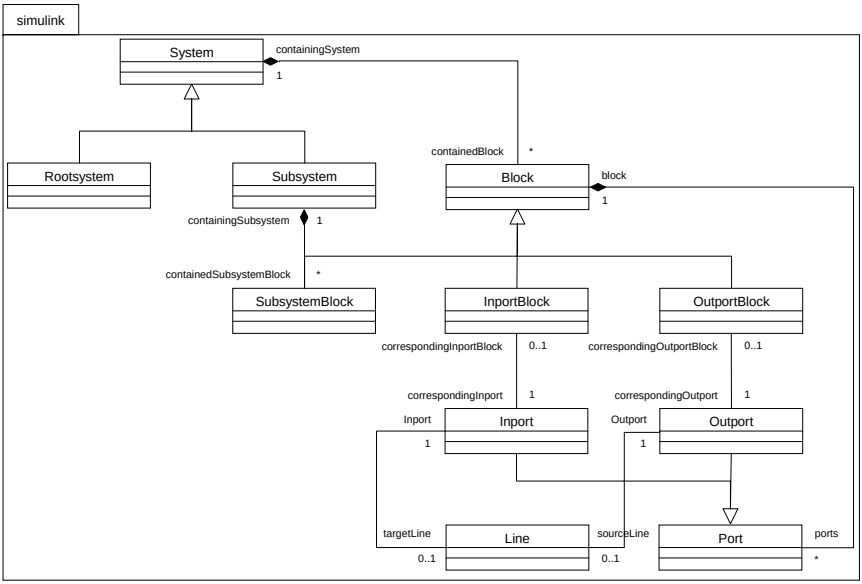

Figure 9.4: Simplified metamodel of Matlab / Simulink

Figure 9.4 depicts a simplified metamodel of Matlab / Simulink. The reader is advised that each element of a Matlab / Simulink model carries a `qualified name`. Basically, a Matlab / Simulink models consists of a `Rootsystem` and a number of `Subsystems`. A `System` (i.e., `Rootsystems` and `Subsystems`) contains `Blocks` (i.e., `SubsystemBlocks`, `InportBlocks`, and `OutportBlocks`). `Subsystems` can be provided with `Inports` and `Outports` which correspond to the `InportBlocks` and `OutportBlocks`. Finally, `Outports` can be connected to `Inports` by `Lines`.

Based on the metamodels presented in Figure 9.3 and Figure 9.4 we come up with the metamodel of our model integration specification (cf. Figure 9.5). We have declared the integration link types `ModelPackageToRootsystem`, `ClassToSubsystemBlock`, `PartToSubsystemBlock`, `InportToInport`, `OutportToOutport`, and `ConnectorToLine`. The integration link types `ClassToSubsystemBlock`, `InportToInport`, and `OutportTo-`

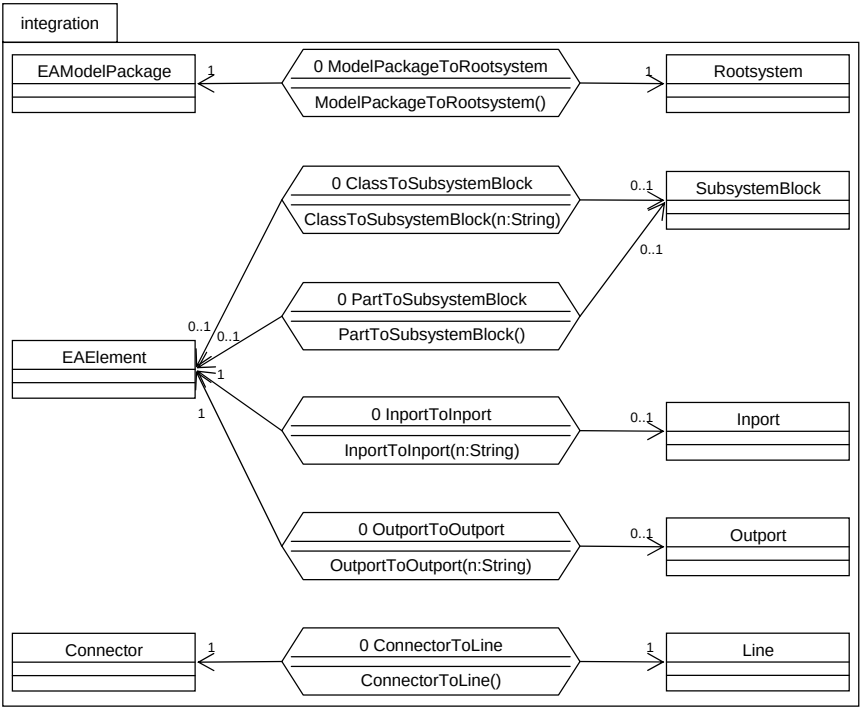

Figure 9.5: Integration metamodel

`Outport` are parametrized, whereas the remaining integration link types are not. Basically, each model element of Enterprise Architect will be linked to one model element of Matlab/Simulink. The multiplicity `{0..1}` indicates *exclusive or*[7]. That means that an `EAElement` will be integrated either with a `SubsystemBlock`, an `Inport`, or an `Outport`. Correspondingly, each `SubsystemBlock` will be integrated with a `EAElement` either by a `Class-ToSubsystemBlock` link or a `PartToSubsystemBlock` link. All declared integration link types are provided with a priority of `0`. As we will see below the patterns of the TGG rules attached to each integration link type ensure that the

[7] Upcoming versions of our TGG editor should provide a syntactic construct for expressing *exclusive or*.

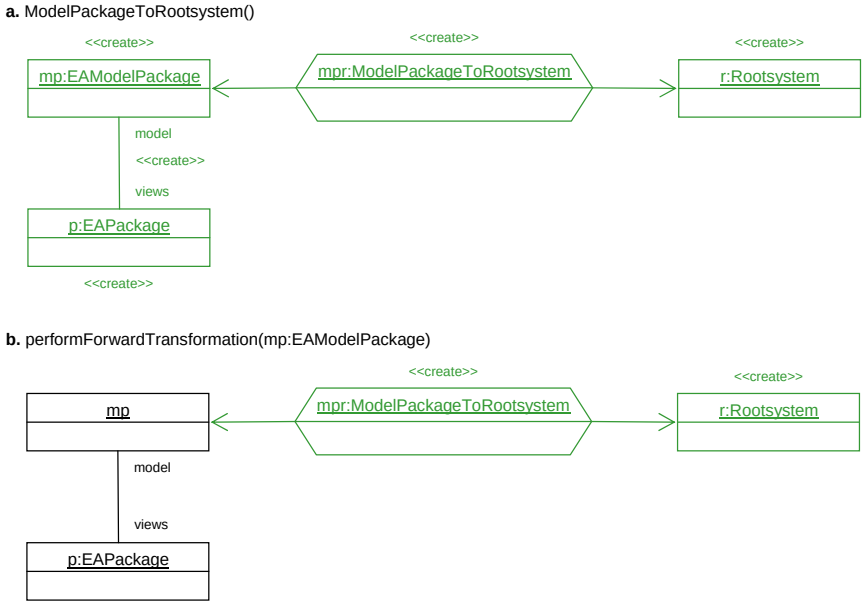

Figure 9.6: `ModelPackageToRootsystem` rule

rules do not conflict with each other (i.e., for each regarded model element at most one rule will be applicable for).

Figure 9.6a. shows the declarative TGG rule of the link integration type `ModelPackageToRootsystem`. The rule has no input parameters and means that the elements `mp` in Enterprise Architect and `r` in Matlab/Simulink are created simultaneously. Furthermore, these elements are linked to each other by a new integration link `mpr`. Finally, a secondary element `p` is created in Enterprise Architect.

Figure 9.6b. shows the operational forward transformation rule which has been derived from the declarative TGG rule in Figure 9.6a. This time the element `mpr` is provided as an input parameter to the rule. The rule creates a new element `r` in Matlab/Simulink and links it to `mp` by a new integration link `mpr` if an element `p` in Enterprise Architect exists which is attached to `mp`.

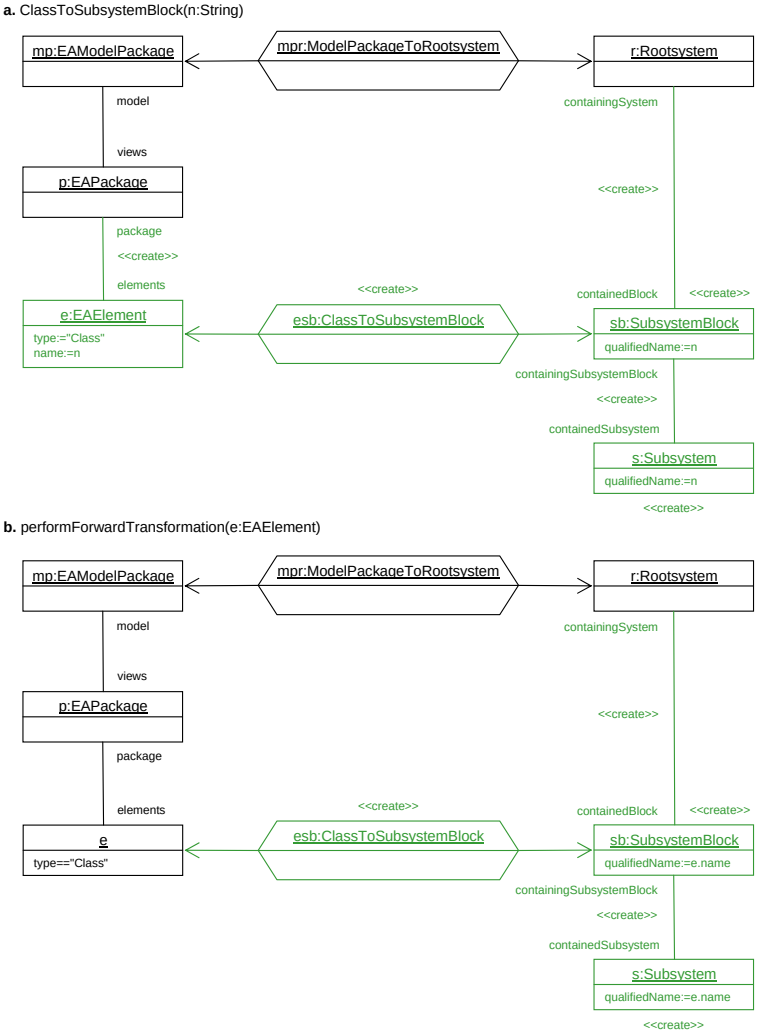

Figure 9.7: `ClassToSubsystemBlock` rule

Figure 9.7a. depicts the declarative TGG rule of the link integration type `ClassToSubsystemBlock`. The rule is provided with an input parameter `n`. The rule simultaneously attaches an element `e` to an existing element `p` in Enterprise Architect and an element `sb` to an existing element `r` in Matalb/Simulink and links them with each other by a new integration link `esb`. Furthermore, the rule creates a secondary element `s` in Matlab/Simulink and attaches it to `sb`. This occurs only if `p` is attached to an element `mp` which is linked by an integration link `mpr` to element `r`.

Again, Figure 9.7b. depicts the corresponding operational forward transformation rule. This rule is provided with an input parameter `e`. The rule attaches a new element `sb` to an element `r` and links it to `e` by a new intergation link `esb`. Furthermore, the rule creates a new element `s` and attaches it to `sb`. This occurs only if `e` is attached to an element `p` which in turn is attached to an element `mp`. Moreover, `mp` has to be linked to `r` by an integration link `mpr`.

Figure 9.8a. presents the declarative TGG rule of the link type `PartToSubsystemBlock`. This rule does not have any input parameters. The rule simultaneously creates a new element `e2` and a new secondary element `c` in Enterprise Architect as well as a new connection between element `s` and `sb2`. Furthermore, the rule creates a new integration link `psb` which links `e2` and `sb2`. This rule is a special case because it does not create a new element in Matlab/Simulink. Rather, it adds a new connection between two already existing elements in Matlab/Simulink only. Additionally, the created integration link also refers to an already existing element in Matlab/Simulink.

The corresponding operational forward transformation rule is presented in Figure 9.8b. This rule links the provided element `e2` with the already existing element `sb2` by a new intergation link `psb`. Thereby, `sb2` has to be already linked to an element `e3` on which `e2` depends on. Furthermore, the rule attaches `sb2` to the element `s` which is contained in an element `sb1` that is linked to the element `e1` to which `e2` is attached to.

The declarative TGG rule of link type `InportToInport` is shown in figure 9.9a[8]. The rule simultaneously adds an element `e2` of the type `Port` and the stereotype `Inport` to Enterprise Architect as well as an element `i` to Matlab/Simulink. Furthermore, `e2` and `i` are linked by a new integration link `ii`. Finally, the rule adds a secondary element `ib` to Matlab/Simulink which correspondes to `i` and is attached to an element `s`. Thereby, `s` is contained in element

[8] The rule of link type `OutportToOutport` looks nearly identical.

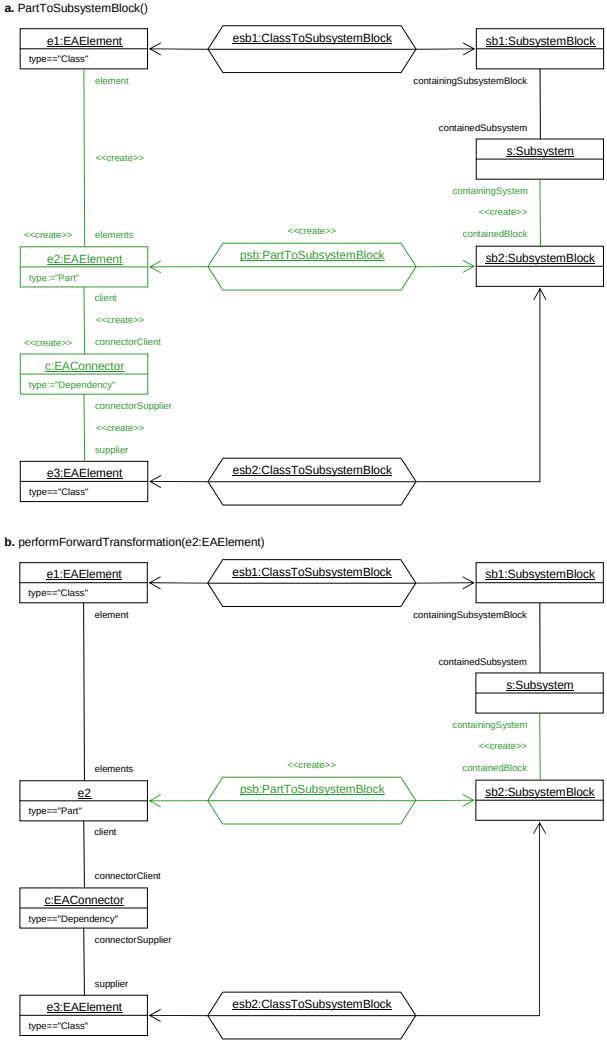

Figure 9.8: `PartToSubsystemBlock` rule

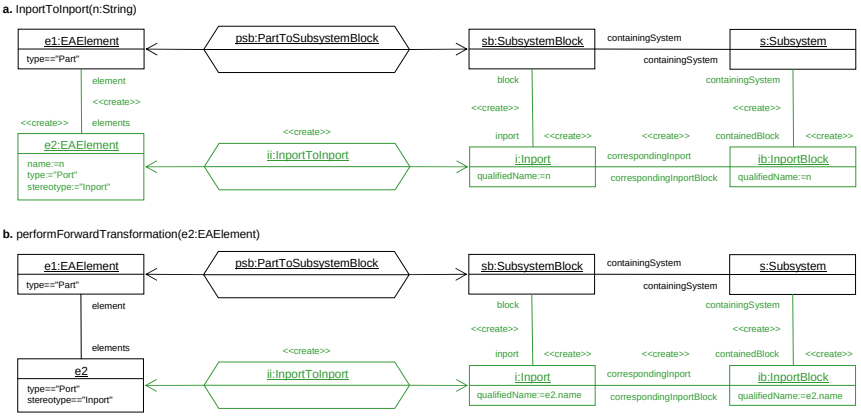

Figure 9.9: `InportToInport` rule

`sb` to which `i` is attached to. `sb` has to be linked by an integration link `psb` to the element `e1` of the type `Part` in Enterprise Architect to which `e2` is attached to.

Figure 9.9b. shows the derived operational forward transformation rule. This rule creats a new element `i` in Matlab/Simulink and links by a link `ii` with the provided element `e2` in Enterprise Architect. Furthermore, `i` is attached to the element `sb` with is linked by a link `psb` with the element `e1` to which `e2` is attached to. Finally, the rule creates a secondary element `ib` which correspondes to element `i` and is attached to the element `s` which in turn is attached to `sb`.

Last but not least Figure 9.10a. depicts the declarative TGG rule of the link type `ConnectorToLine`. This rule deals with the simultaneous creation of a `Connector` in Enterprise Architect and a `Line` in Matlab/Simulink. A `Connector` in Enterprise Architect connects two elements (e.g. `EAElements` with type `Port` and stereotypes `Inport` and `Outport`. Similarly, a `Line` in Matlab/Simulink connects `Outports` with `Inports`.

The operational forward transformation rule which is depicted in Figure 9.10b. transforms the provided element `c` which connects the elements `e1` and `e2` into an element `l` which connects the elements `i` and `o` which correspond to `e1` and

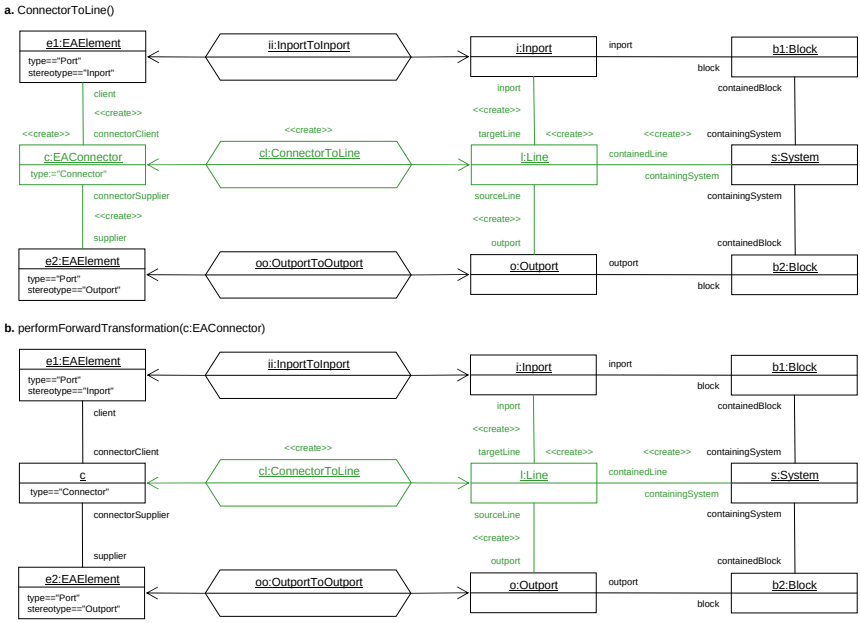

Figure 9.10: `ConnectorToLine` rule

`e2` respectively. Thereby, `l` will be attached to the element `s` which owns the elements `b1` and `b2` which are attached to `i` and `o` respectively.

10 Related work

In this chapter we compare our model integration approach with various related approaches in order to discuss strengths and flaws. To this end we come up with a number of categorization criteria we use to benchmark all approaches. We begin our benchmark with the discussion of approaches that deal with model integration in general but use entirely different approaches than we do. We then continue with approaches that directly utilize operational graph grammars. After that we examine related triple graph grammar approaches that yet usually do not aim at compliance to the QVT standard. Finally, we evaluate approaches that aim at compliance to (parts of) the QVT standard but do not utilize (triple) graph grammars.

10.1 Categorization criteria

In order to compare the different approaches with each other we evaluate to which extend the criteria provided by the QVT-RFP as outlined in Section 3.2 have been implemented. Since only QVT-based approaches intentionally aim at compliance to the QVT requirements it would not be surprising when the other approaches do not adhere to these requirements. Therefore, only using the criteria provided by the QVT-RFP does not appear to be fair.

[CH03] provides a more objective and overarching set of criteria which can be used to evaluate any model transformation approach independently from the QVT standard. At top-level [CH03] considers the following categories: *Transformation Rules*, *Rule Application Scoping*, *Source-Target Relationship*, *Rule Application Strategy*, *Rule Application Strategy*, *Rule Scheduling*, *Rule Organization*, *Tracing*, and *Directionality*.

Transformation Rules

This category examines in which way model transformation rules are written down in a considered approach. [CH03] assumes that rules are composed of a

left-hand and a right-hand side. In contrast to graph rewriting systems where the left-hand side of a rule refers to the part of a rule that has to be matched in the host graph and the right-hand side designates the modification to the host graph [CH03] appoints that the left-hand side (LHS) refers to the source model and the right-hand side (RHS) refers to the target model. However, LHS and RHS may be provided with patterns or logic expressions. Patterns can be written as strings, terms, or graphs. The syntax in which patterns are written down can meet the abstract or the concrete syntax in which source and target models are written in. Elements of the patterns can be typed or untyped. Logic expressions can be executable (imperatively or declaratively) or non-executable. Additionally, patterns can involve (typed or untyped) variables.

Rules can contain the LHS and the RHS separately or in a collapsed style. Furthermore, rules may be executable bidirectionally or unidirectionally only. Rules might be parameterizable or nor. Some approaches need to construct intermediate structures that neither belong to the source nor to the target model and usually are dropped at the end of the model transformation.

Rule Application Scoping

This category distinguishes whether the scope of a model transformation can be reduced with respect to the source and the target model (i.e., only parts of the models are considered).

Source-Target Relationship

This category discusses in which way a target model is constructed from a given source model. Each model transformation approach usually is able to create a new target model from a source model (i.e., a batch transformation). In addition an approach may be able to modify an already existing target model (i.e., an incremental transformation). The case where source and target model coincide is called an in-place transformation. If an approach is able to modify an existing target model [CH03] distinguishes whether the transformation may only extend the target model or additionally may delete existing elements.

Rule Application Strategy

Usually there are multiple locations in the source model to which a given rule can be applied to. The transformation needs to select the locations to which the rule actually should be applied to. Basically, the algorithm to select these locations can be deterministic, non-deterministic, or interactive.

Rule Scheduling

This category introduces criteria concerning the order in which the transformation rules are applied. First of all, the scheduling of rules is implicit or explicit. Implicit rule scheduling means that the user has no means to directly influence the order in which rules are applied. In contrast explicit rule scheduling means that the user is able to directly influence this order. Explicit rule scheduling is called external if the approach provides control structures that are clearly separated from the rules. Explicit rule scheduling is called internal if the approach allows rules to directly invoke other rules.

Furthermore, the transformation algorithm needs a strategy to select a rule that is to be applied next. Again this strategy can be deterministic, non-deterministic, or interactive. Additionally, the strategy needs to be able to resolve conflicts (i.e., multiple rules are applicable at the same time).

Moreover, an approach might be able to apply rules iteratively, recursively, or based on fixpoint iteration. Finally, the transformation process may be subdivided into several distinct phases (e.g. transform the structure of the source model first, deal with the attribute values after).

Rule Organization

This category discusses in which ways a set of rule can be organized. This includes means for modularization (i.e., package concepts) and reusability (i.e., inheritance and rule composition). Finally, the set of rules can be organized according either to the structure of the source or the target model, or independently from them.

Tracing

Approaches may provide dedicated support for traceability links (i.e., links that materialize the dependencies between source and target model elements). [CH03] distinguishes approaches that store traceability links either in the source or the target model, or separately from them. Furthermore, the creation of rules can be controlled either manually or automatically for all or just for some rules.

Directionality

The last category deals with the direction in which a set of rules is applicable. Rules of unidirectional approaches can only be applied from source to target. These approaches need an entirely new set of rules for performing a transformation in the reverse direction. In contrast bidirectional approaches can apply one set of rules in both directions. Either the rules themselves are applicable in both directions or the set of rules contains complementary unidirectional rules[1].

10.2 Common approaches

In this section we examine common (model) integration approaches that use an entirely different approach or have a different focus than we do. Particularly, we discuss data warehouses, federated databases, and enterprise application integration solutions.

Data Warehouses

Basically, a *data warehouse* [KR02] is a database to which data from operational systems is propagated to in order to perform time consuming integrated analysis on the data. When the data is available at the database the desired analysis can be expressed using the query language of the database. The challenge is to create a database schema that is capable of storing the data of all considered operational systems. Usually, every time a new system is added the schema of the data warehouse has to be adjusted. On the one hand this task is very intricate, on the other hand the task becomes even harder when the number of operational systems grows.

[1] The author is the opinion that such approaches should not be considered bidirectional.

Federated Databases

In a *federated database system* [HM85] a component called mediator is used to access a number of databases of operational systems through one common interface. For a client it looks like it is accessing one integrated database. In order to calculate the result of a query the mediator needs to decompose the query into subqueries that are to be performed on each affected database. The results of each subquery need to be merged by the mediator in order to provide the result of the actual query. As pointed out in [Zha94] one crucial point in federated databases is the coordination or integration of the schemas of the regarded databases into one common database schema. Thus, federated database systems suffer from the same problems as data warehouses do.

Enterprise Application Integration

Enterprise Application Integration (EAI) [Gab02] deals with the integration of heterogeneous applications by connecting them with each other by means of adapters. Thereby, integration can happen on different levels such as *data level*, *process level*, or *user-interface level*. Furthermore, there are at least three possibilities of how to connect applications to each other. First of all, applications can be connected to each other *point-to-point*. Secondly, applications can be connected to each other in a *hub-and-spoke* architecture. Finally, applications can be connected to a common *application integration bus* (e.g. SAP Netweaver[2]). In order to avoid that each application must transform its data into the format of every other connected application, usually a common data exchange format (i.e., a schema in terms of database systems) is defined. Therefore, EAI approaches suffer from the same problems as data warehouses and federated database systems do.

As data warehouse solutions, federated database, or EAI solutions usually are realized by manually implementing the desired integration aspects neither the requirements of the QVT-RFP nor the categorization criteria from Section 10.1 can reasonably be applied. Furthermore, the manual implementation of a model integration cannot be regarded as a model integration approach at all. Rather, we suggest the application of model integration approach such as QVT or TGGs as introduced in this work.

[2] http://www.sap.com/platform/netweaver/index.epx

10.3 Graph Grammar-based approaches

The approaches presented in this section rely on graph grammars respectively graph transformations as introduced in Chapter 4. Our own approach uses graph transformations as an intermediate language between specification and executable code as pointed out in Chapter 7.

Among others *PROGRES* [Sch91], *FUJABA* [Zün01], *MOFLON*[3] [Ame08], *AGG* [TB94], and *GReAT* [BNvBK06] are typical representatives of graph grammar / transformation-based systems. *AToM*^{3} [dLV02] utilizes graph transformations for the definition of views.

Concerning the categorization we focus on MOFLON as a representative of graph grammar / transformation-based systems where suitable as it is designed to meet OMG's world of meta modeling. Regarding the general requirements of OMG's QVT-RFP (cf. Section 3.2) we state that MOFLON's schema part of a graph transformation specification relies on the MOF 2.0 metamodel. However, the rule part is adopted from the underlying FUJABA system. FUJABA entirely relies on a UML 1.x-like metamodel. On the one hand we have discovered that the impact of the new MOF 2.0 features on the rule level are negligible [Ame08]. On the other hand there are plans to merge FUJABA and MOFLON and relying on MOF 2.0 as the underlying metamodel for rules as well.

Considering the mandatory requirements of OMG's QVT-RFP we see that the left-hand sides of graph transformations rules act as queries whereas the right-hand sides express the modifications (i.e., transformations). Altogether, graph transformations are suitable for expressing queries, views, and transformations. As already mentioned above their drawbacks are that queries, views, and transformations have to be expressed operationally rather than declaratively. Furthermore, it is not possible to explicitly keep source, target, and tracing model separate from each other. In principle it is possible to realize incremental model transformations with graph transformations. However, it is a matter of the utilized rule execution engine.

With respect to the optional requirement of OMG's QVT-RFP we point out that graph transformations cannot be used to express bidirectional model transformations. Rather, each direction has to be specified explicitly. Moreover, MOFLON

[3] disregarding our TGG implementation

incorporates a number of concepts for reusing and extending existing model transformations [Ame08]. Finally, graph transformations are suitable for expressing in-place model transformations; in fact they are designed for this purpose.

Looking at the categorization criteria of [CH03] we additionally find that graph transformations do not separate the left-hand side (i.e., the source model in terms of [CH03]) from the right-hand side (i.e., the target model in terms of [CH03]). Usually, graph transformations rely on parametrized graph patterns which can be assigned with executable logic expressions. To the best of our knowledge no graph transformation system offers the possibility to reduce the scope of a graph transformation. As graph transformations do not explicitly support the separation of source and target models they actually realize in-place transformations only. Depending on the regarded graph transformation system any type of rule application strategy can be supported. Systems as PROGRES or FUJABA / MOFLON usually rely on a non-deterministic rule application strategy. Similarly, graph transformation systems realizes any type of rule scheduling. Usually, graph transformation systems (particularly PROGRES and MOFLON) provide means for rule organization as package concepts and means for reusing existing graph transformations. Graph transformation systems are capable of supporting traceability links provided that they are explicitly modeled and regarded as part of the in-place transformation. Thus, traceability links are not separated from the source and target model parts. Finally, graph transformation rules are applicable in one direction only. If the other direction is needed it has to be specified explicitly and is not derived automatically.

10.4 TGG-based approaches

The approaches presented in this section rely on the initial idea of triple graph grammars as pointed out in Section 4.5. Particularly, we focus on the approaches of [Bec08], [Wag08], and [GK07]. Our own approach belongs the the category of these approaches. However, for categorization purposes we disregard our own approach for a moment in order to be able to emphasize the contributions of our approach.

Concerning the general requirements of OMG's QVT-RFP we find that none of the currently available TGG approaches conforms to the MOF 2.0 standard. For writing TGG rules [Bec08] relies on a UML / MOF 1.4 metamodel which has been extended by own concepts. From the specified TGG rules [Bec08] derives

a PROGRES specification which contains the corresponding operational rules. PROGRES itself does not conform to any of the OMG standards. [Wag08] and [GK07] rely on FUJABA for the creation of TGG specifications. For each declarative specification [Wag08] and [GK07] derive another FUJABA specification which contains the operational rules. FUJABA's metamodel conforms more or less to the UML 1.x metamodel.

Regarding the mandatory requirements of OMG's QVT-RFP we state that all TGG approaches are suitable for the specification of queries, views, and transformations. However, [Bec08], [Wag08], and [GK07] do not integrate models that conform to the MOF 2.0 standard. Again, they rely on the out-dated UML 1.4 standard.

Looking at the optional requirements of OMG's QVT-RFP we find that all TGG approaches allow for the declarative specification of bidirectional model-to-model transformations. As each TGG approach relies on the explicit separation of source, target, and trace model they are not suitable for in-place transformations. For this purpose the user has to utilize the underlying single graph transformation language. Neither [Bec08], [Wag08], nor [GK07] regard means for the modularization or reuse of model-to-model transformation specifications.

Taking the categorization criteria of [CH03] into account we see that all TGG approaches explicitly separate the LHS from the RHS of a regarded model-to-model integration rule. Each rule is written as a graph transformation rule and, therefore, contains typed variables. All TGG approaches support the usage of executable logic expressions. In contrast parametrized TGG rules are not supported. Neither [Bec08], [Wag08], nor [GK07] allow for the reduction of the scope of a model-to-model integration. The TGG approaches of [Bec08], [Wag08], and [GK07] perform model-to-model integrations in a pseudo-incremental manner (i.e., they perform the integration batch-oriented but regard the results of preceding integration runs). Concerning the rule scheduling aspect [Bec08], [Wag08], and [GK07] rely on a non-deterministic rule scheduling strategy. For resolving conflicts [Bec08] asks the user which rule is to be applied next.

10.5 QVT-based approaches

In order to categorize QVT-based model integration approaches we can separate these approaches with respect to the part(s) of QVT (i.e., relational, core, or operational) which a regarded approach aims at. Furthermore, we can separate the

approaches from each other with respect to the type of notation (i.e., visual or textual) that is provided by the approaches.

Approaches that address the operational part of QVT only are not considered as related work for this thesis. Approaches like Borland Together[4], SmartQVT[5] from the Modelware project, and XMF Mosaic[6] belong to this category.

Approaches such as Medini[7] and ATL [JK05] aim at the relational[8] part of QVT but provide a textual notation only. Approaches such as Model Morf[9] aim at the relational part of QVT and provide a graphical notation as well. However, in this thesis we regard any approach which aims at the relational part of QVT as related work.

Having already introduced the relational part of QVT in detail in Section 3.3.2 we now just summarize the properties of this part of QVT with respect to the requirements of OMG's QVT-RFP and the categorization criteria of [CH03]. Concerning the general requirements of OMG's QVT-RFP we find that the QVT standard relies on and extends the MOF 2.0 metamodel. Looking at the mandatory requirements of OMG's QVT-RFP we see that the QVT standard provides sublanguages for the specification of queries and transformations. However, the creation of views is intentionally disregarded, yet. Naturally, the QVT standard supports the integration of MOF 2.0-compliant models. Regarding the optional requirements of OMG's QVT-RFP we can state that the relational part of the QVT standard allows for the declarative and bidirectional integration of models. The QVT standard does not explicitly list means for the modularization and reuse of model integration specifications. As mentioned in Section 3.3.2 many questions concerning the metamodel of QVT are left open. Moreover, the QVT standard implicitly maintains traceability information for the models being integrated with each other. Finally, the QVT standard allows for the specification of in-place transformations.

Regarding the categorization of [CH03] we see that the relational part of QVT provides separate LHS and RHS for the model integration rules. QVT supports

[4] http://www.borland.com/de/customers/informs/16_dez_05/together_2006.html
[5] http://universalis.elibel.tm.fr/qvt/
[6] http://www.xactium.com
[7] http://www.ikv.de/index.php?option=com_content&task=view&id=75&Itemid=77
[8] In fact ATL aims at the core part of QVT only.
[9] http://www.tcs-trddc.com/ModelMorf/index.htm

	Common approaches	Graph Grammar-based approaches	TGG-based approaches	QVT-based approaches	Our approach
Formalism	-	+	+	-	+
Semantics	-	+	+	-	+
Standard-compliance	-	(+)	-	+	+
Acceptance	+	-	-	+	+
Usability	-	-	(+)	+	+

Figure 10.1: Comparison of various model integration approaches

patterns (graphically and textually notated) as well as logic expressions and typed variables. As model integration rules in the relational part of QVT are declarative the rules can be applied bidirectionally. The QVT standard does not discuss any possibility for reducing the application scope of a given model integration specification. The QVT standard does not demand a certain strategy (e.g. batch-oriented or incremental) for the transformation of a source model into a target model. The standard only provides declarative specifications. The actual application strategy is left to the regarded implementation of the standard. The QVT standard claims that it is suitable for specifying in-place model transformation (i.e., source and target model coincide) but it is obvious that the standard is not dedicated to this scenario. As the QVT standard is based upon the MOF 2.0 standard it might be able to adopt the package concepts of MOF 2.0 for the organization of rules. However, the QVT standard does not discuss this issue at all. Finally, a declarative QVT specification implicitly maintains a set of correspondence links that can be used for traceability purposes.

10.6 Summary

Figure 10.1 strikingly summarizes our comparison of the various model integration approaches presented in this chapter and our own approach. We claim that our approach combines the strengths of TGGs with the strengths of QVT and,

therefore, addresses the aspects of formalism and semantics as well as the aspects of standard-compliance, acceptance, and usability.

11 Conclusion

This chapter concludes this thesis and summarizes the results and contributions of our approach. Furthermore, this chapter points out open issues and discusses future work.

In this thesis we have explained our model integration approach that combines the more formal and theoretical concept of *Triple Graph Grammars* (TGGs) with the more practically oriented OMG standard *Query / View / Transformation* (QVT). By doing so we, hopefully, have been able to compensate the flaws of each approach with the strengths of the other. Particularly, by aiming at the QVT standard we want to increase the acceptance of our approach in practice. By relying on the formally well-defined and mature TGGs we want to compensate QVT's lack of a proper formal foundation which makes it hard to implement and apply this standard.

Compared to the original TGG approach as presented in [Sch94] our approach relies on recent (OMG) standards and, thus, hopefully has a higher acceptance. Furthermore, our approach adds a number of user-friendly features not only from the QVT standard. Finally, our approach provides more sophisticated rule derivation and application strategies and addresses a number of issues that are open in the original TGG approach. Compared to the original QVT standard as described in [OMG05b] our approach relies on a more precise metamodel. Furthermore, our approach relies on the well-known formalism of TGGs and, thus, has a more precise semantics. Finally, our approach adds a number of useful features that are not included in the QVT standard.

We started by pointing out the increasing relevance and acceptance of models in the industrial area. We then motivated the pestering need for (semi)-automatic model integration solutions; i.e., detecting inconsistencies between models and taking recovery measures.

After that we presented an introduction to OMG's world of model related standards. Particularly, we introduced *Model Driven Application Development* (MDA) as OMG's vision of software and system development. OMG's *Meta Object Facility* (MOF) allows for the graphical specification of modeling languages

and metamodels. OMG's *Object Constraint Language* (OCL) complements MOF with textual constraints. Finally, we explained OMG's *Query / View / Transformation* (QVT) standard which aims at the declarative and operational rule-based specification of model integration scenarios.

Thereafter, we presented the non-standard-compliant world of graph grammars which already has been transferred to OMG's world of models beforehand. Starting with well-known string grammars and ordinary graph grammars we concluded with pair grammars as proposed by [Pra71] in the 70s and the initial ideas on triple graph grammars from [Sch94] in the 90s.

Subsequently, we described our own approach which is based on TGGs and extends them by concepts from OMG's QVT standard. The result is a model integration language which combines user-friendly constructs from QVT with the formal foundation of TGGs. In contrast to QVT which relies on EMOF only our language relies on complete MOF 2.0 which provides more sophisticated concepts (e.g. refinement, redefinition, merge). Like TGGs our language consists of a schema part and a rule part. The schema part defines the structure of a model integration scenario and provides sophisticated means for modularization and reuse. The rule part declaratively specifies the intended behavior of the model integration scenario and provides sophisticated means for rule conflict resolution.

Next, we presented how we derive operational rules from a given declarative rule set. These operational rules can be applied in order to check for and recover consistency of two models. Additionally, we introduced strategies how to apply these rules for realizing the intended model integration.

Moreover, we explained how we implemented our approach as part of the meta-CASE tool MOFLON. Using the MOFLON-TGG plug-in a user can create a declarative model integration specification from which MOFLON can generate executable Java code. This Java code is dynamically be loaded into our integration framework which applies the operational model integration rules represented by the Java code.

Referring to a number of CASE studies we proved that our approach actually is useful in practice. Particularly, we demonstrated the application of our approach for the model integration scenarios traceability link creation, model-model consistency analysis, and model-model transformation.

Finally, we compared our proposal with related approaches. Thereby, we considered entirely different approaches, graph grammar-based approaches, TGG-based approaches, and recent implementations (of parts) of the QVT standard.

11.1 Open issues

Concerning our approach and its current implementation we are facing two major issues.

Incremental transformation

First of all, the derived operational rules[1] as described in Chapter 7 can only be applied in a batch-oriented manner. The reason is that those rules cannot identify already existing model elements that should be reused. Rather, the presented rules create new model elements each time they are invoked. The underlying challenge is to decide whether two given references to model elements refer to the same model element. To this end we need to define a key concept as defined by the QVT standard for instance (cf. Chapter 3). As we already pointed out in Chapter 3 QVT's key concept is not satisfying. Rather, a new precisely defined key concept should be added to the MOF standard.

Delaying model elements

Moreover, our current implementation of the model integration framework that applies the operational model integration rules is not able to delay the integration of a given model element as presented in Section 7.2. The reason is that we need a concept of keeping track of already integrated model elements. A naive approach would be to add an additional field `is_integrated` to all types of model elements. We strongly advise against this possibility because the `is_integrated` field merely is a technical field that should not appear in any models' metamodel. Furthermore, this possibility is not applicable to cases where the metamodels are already given and immutable.

Another possibility is to apply the *Decorator Pattern* [Gra02]. The basic idea is that a decorator wraps a concrete object and adds additional functionality. Again we would add an additional field `is_integrated`. As this field resides in the decorator we do not undesirably modify any metamodels. The major drawback is that the decorator must implement the same interface as the wrapped object.

[1] At least those rules that create new model elements.

Since JMI defines interfaces with many methods the creation of the needed decorators is only feasible if the decorators can be automatically generated. The implementation of such a generator that analyzes given JMI-compliant metamodels and generates the desired decorators is a challenging task that is out of scope for this thesis.

Finally, one possibility is to extend the derived operational rules by the needed management functionality that keeps track of which elements have already been integrated and which not. The drawback is that we again are mixing the technical and conceptual space. This time this is admissible since the operational rules are automatically generated and can be considered as intermediate code. The user's specification remains unchanged. We propose to implement this solution in the next release of MOFLON's TGG plug-in.

11.2 Future work

Based on our model integration approach as presented in this thesis there are a number of further extensions that could be made.

Model copying concept for TGGs

Stratification as proposed by [KKG05] is an approach that allows for the specification of software systems on different layers of abstraction. Basically, the idea is that the model of a software system (e.g. a class diagram) can be enriched by *Annotations*. Such an annotation can be used to express that a certain design pattern (e.g. observer pattern) should be applied. Thus, each annotation compactly represents a certain structure (i.e., classes, attributes, and associations) that is added to the system without showing unnecessary details. On the most abstract level the modeler sees all annotations that have been added to the model. From the most abstract level the modeler switches to a more concrete level by *unfolding* annotations. On the most concrete level the model does not contain any folded annotations. The result of unfolding a number of annotations may depend on the order in which the annotations are unfolded.

Ideally, the modeler can arbitrarily fold and unfold annotations at any time. Furthermore, the modeler should be able to arbitrarily swap levels of abstraction. Rather than deriving a new level of abstraction each time the user folds or unfolds

an annotation all already derived levels of abstraction should concurrently exist and automatically be kept consistent with each other. Thus, we do not deal with only one model which is subject to an in-place model transformation when folding or unfolding an annotation. Rather, we have to deal with multiple models that are quite similar to each other that have to be kept consistent by means of incremental updates with each other.

Our idea is to integrate all concerned levels of abstraction by linking them pairwise by applying TGGs. The drawback is that two levels of abstraction may be very similar to each other and only differ in the place where an annotation has been unfolded. Thus, most rules of a TGG specification deal with just copying one model to an identical clone. The specification of those straight-forward model copying rules is very cumbersome. Therefore, it would be very convenient if the TGG modeler only has to specify such rules that realize the folding and unfolding of annotations.

View Triple Graph Grammars

One issue that the current QVT standard intentionally disregards is the definition of views as they are already known in database systems. A *View* is helpful in the following situation. A user wants to explore and manipulate a model. The model presents the desired information in a way that is not well-suited for the user's intention. For instance the model complies to a metamodel that is too technical or contains too many unnecessary details from the user's point of view. A view is a model that corresponds to the underlying (base) model the user wants to deal with. The view presents the desired information in a way that is well-suited for the user's intention. To this end the view model complies to a metamodel that is well-suited for the user's needs. Basically, there a two possibilities how to realize such a view model.

Firstly, the view model can be *materialized*. That means that there actually is a model that complies to the view's metamodel. The view model is created from the base model by means of model integration. Changes to the view or the base model must be propagated to the other model and vice versa. As a materialized view more or less is a copy[2] of the base model the view model requires roughly the same amount of space as the base model.

[2] Actually, the view model can contain less details or derived information.

Secondly, the view model can be realized in a virtual manner. That means that the user still operates directly on the base model but uses interfaces that are well-suited for their needs. As both models coincide changes to the base model or the virtually existing view model are implicitly "propagated" to the "other" model. As there actually exists the base model only the view model does not require additional space. In exchange, manipulating the base model through the view interfaces requires additional time. Furthermore, implementing adapters that map the view interfaces to the base model is not trivial and might be impossible when view and base metamodel are too different.

In order to realize materialized views we can just apply our own TGG approach as presented in this thesis. For realizing virtually existing views we proposed an extension to our TGG approach called *View Triple Graph Grammars* (VTGGs) in [JKS06]. Basically, the idea is that the VTGG specification that declaratively specifies the simultaneous evolution of both view and base model is implemented by means of class adapters (cf. [Gra02]). Thereby, each class adapter maps an interface of the view metamodel to an interface of the base metamodel. To this end the class adapter inherits from an implementation of a base metamodel interface and delegates method calls to the view interface to this implementation. In [JKS06] we implemented these class adapters manually. Needless to say that such class adapters should be generated automatically. Currently, our proposal merely is an initial idea. There are a number of issues which must be dealt with in detail. For instance the manual implementation or generation of the needed class adapters becomes intricate when two or more concurrently existing views should be realized. In this case each class adapter has to implemented multiple interfaces at the same time. The situation becomes even worse when views should be based on top of other views rather than on directly the base model.

Multi Domain Integration

When dealing with software and system development processes the number of involved documents and models is much large than just two. Usually, the users need automatic support for keeping all documents consistent with each other. Generally, it might be possible to realize the desired consistency by keeping the involved models pairwise consistent with each other. For each pair of models the users can specify one TGG specification using our approach as presented in this thesis. The major drawback is that you need at least `n-1` TGG specifications for keeping `n` documents pairwise consistent with each other. Moreover, it might be the case

that one model does not only depend on the information stored in another model. Rather, the model may depend on the information stored in an arbitrary number of other models.

Therefore, we presented an extension to our TGG approach in [KS06] that allows for the declarative integration of an arbitrary number of involved models. Ideally, for integrating n documents you need to specify only one declarative *Multi Domain Integration* specification. From this specification the desired operational model integration rules can be derived automatically similar to the strategy described in Chapter 7.

The major drawbacks are that one declarative specification that integrates all involved models at the same time will be very large and intricate. Additionally, the derivation of operational rules will be much more complex.

11.3 Closing words

In this thesis we have presented a model integration language which combines well-known Triple Graph Grammars with the upcoming model integration standard QVT from the OMG. As a result we provided one of the first implementations of the graphically notated declarative part of QVT, which is based on the proper formal foundation of graph transformations.

Nevertheless, we have shown that there still are a number of open issues and possibilities for further extensions to TGGs in general and our approach in particular that should be made in the future. Currently, TGGs, further extensions, and their relationship to QVT still are under investigation at the universities of Aachen, Paderborn, Kassel, Berlin, Marburg, and Darmstadt.

A Running example

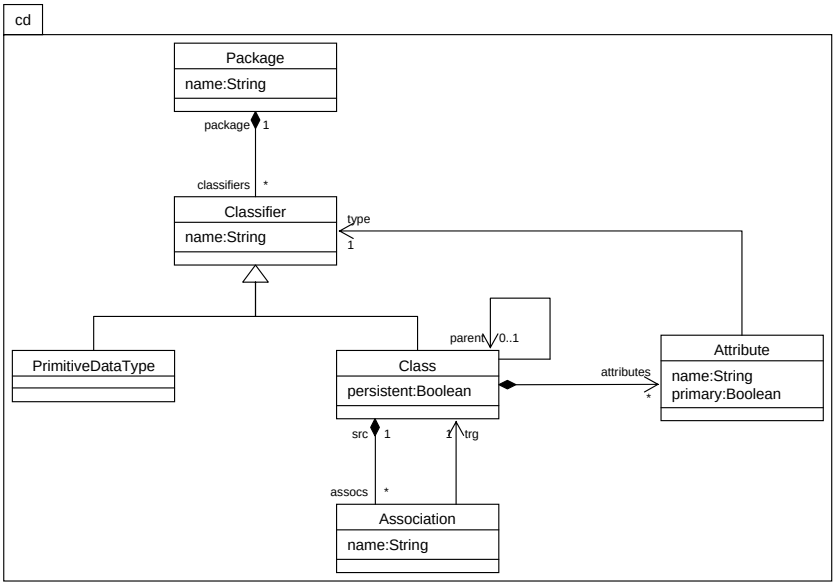

Figure A.1: Metamodel for class diagrams

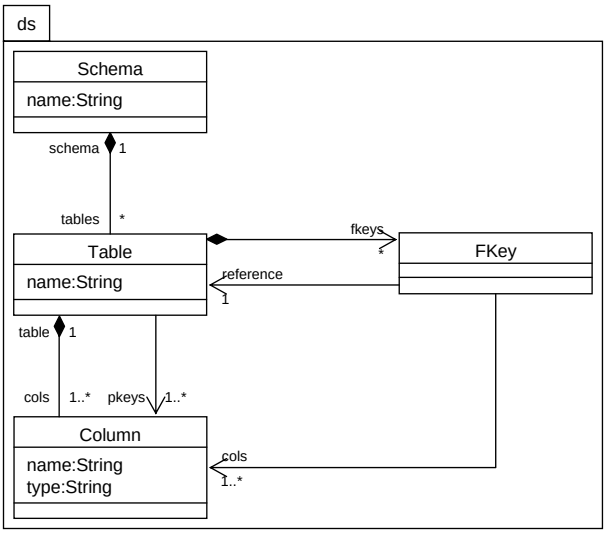

Figure A.2: Metamodel for database schemas

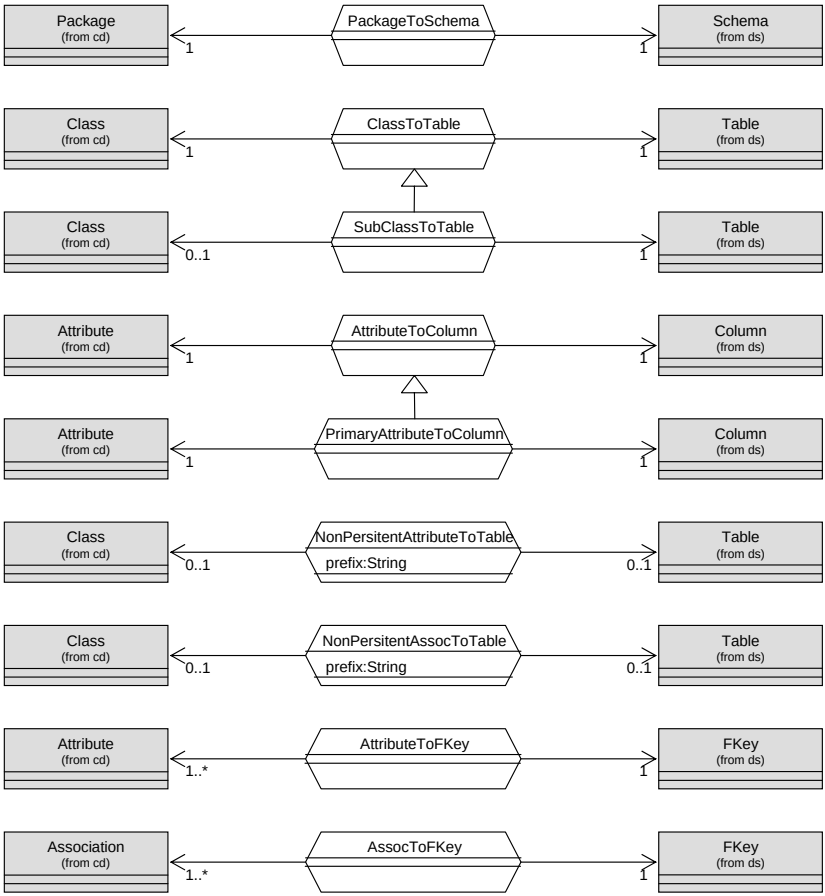

Figure A.3: Integration metamodel

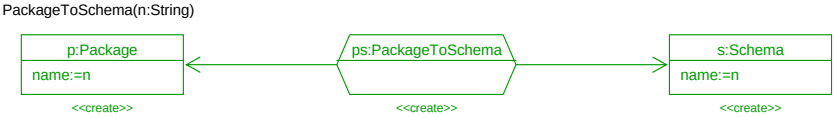

Figure A.4: TGG rule `PackageToSchema`

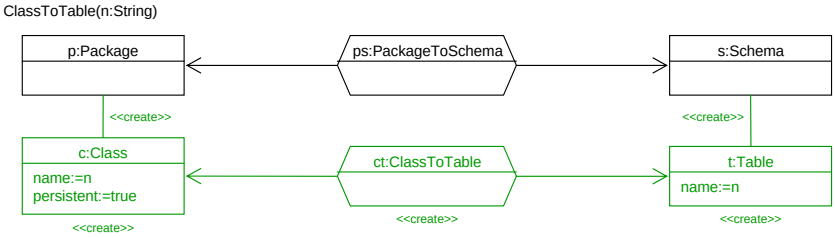

Figure A.5: TGG rule `ClassToTable`

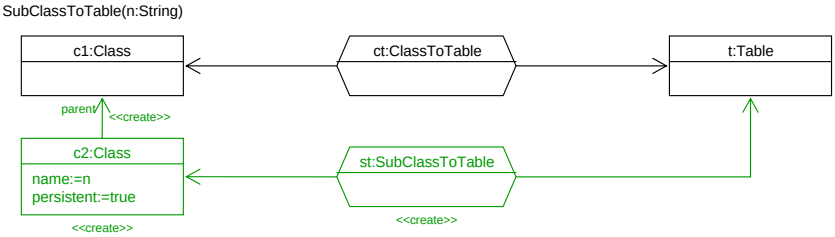

Figure A.6: TGG rule `SubClassToTable`

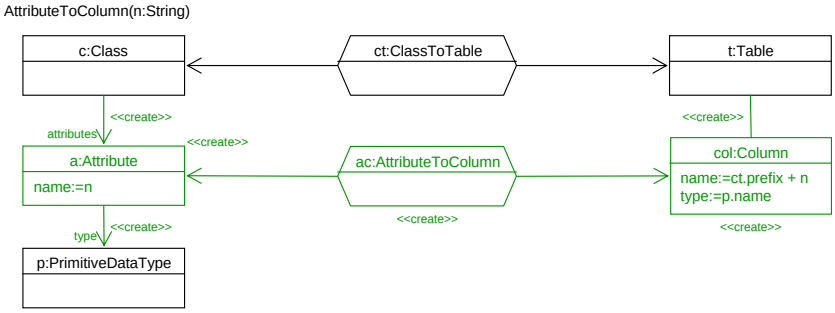

Figure A.7: TGG rule `AttributeToColumn`

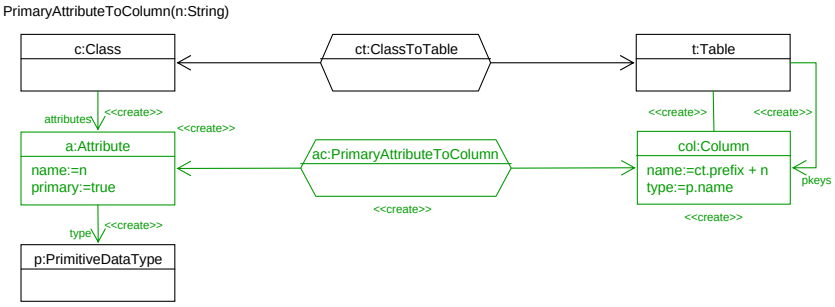

Figure A.8: TGG rule `PrimaryAttributeToColumn`

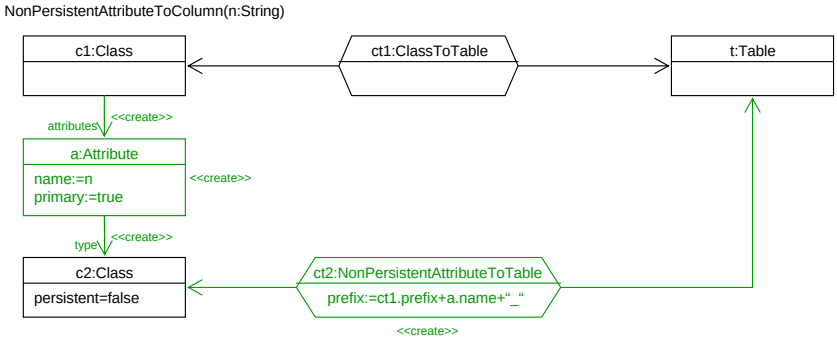

Figure A.9: TGG rule `NonPersistentAttributeToColumn`

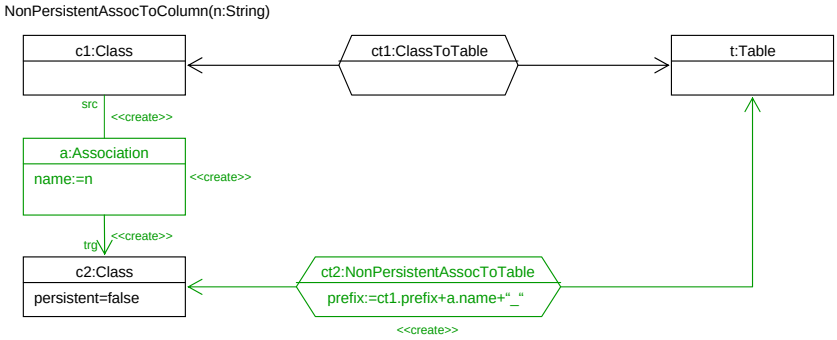

Figure A.10: TGG rule `NonPersistentAssocToColumn`

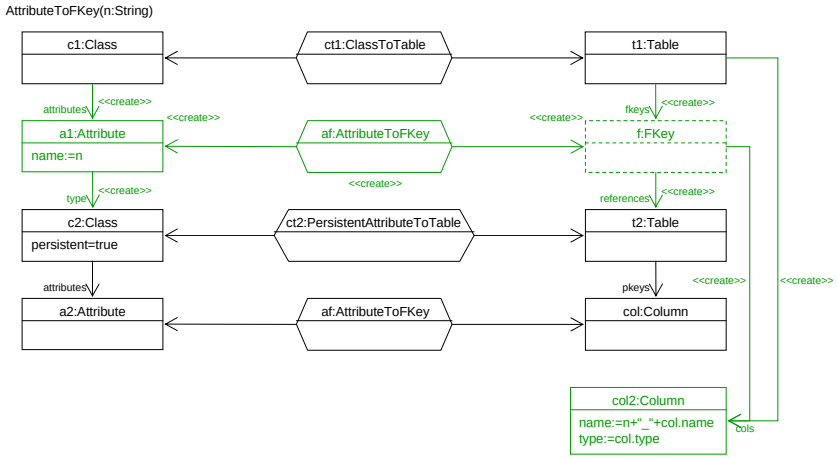

Figure A.11: TGG rule `AttributeToFKey`

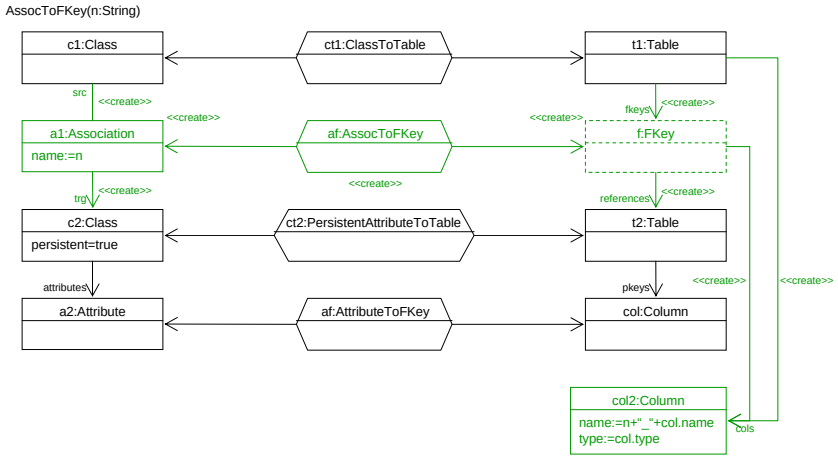

Figure A.12: TGG rule `AssocToFKey`

B Curriculum vitae

since 08/07	Software engineer at sd&m AG in Düsseldorf
08/02 – 07/07	Research assistent at TU Darmstadt
09/96 – 07/02	Studies of Computer Science at RWTH Aachen
09/95 – 09/96	Civilian service at St. Josef nursing home in Krefeld
06/95	Abitur at Gymnasium Thomaeum in Kempen

Bibliography

[ABS04] C. Amelunxen, L. Bichler, and A. Schürr. Codegenerierung für Assoziationen in MOF 2.0. In *Proceedings Modellierung 2004*, volume P-45 of *Lecture Notes in Informatics*, pages 149–168, Bonn, 3 2004. Gesellschaft für Informatik.

[ADS02] F. Altheide, H. Dörr, and A. Schürr. Requirements to a Framework for Sustainable Integration of System Development Tools. In *Proc. 3rd European Systems Engineering Conference*, 2002.

[AKRS03] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. Adapting FUJABA for Building a Meta Modelling Framework. In H. Giese and A. Zündorf, editors, *Proc. FUJABA Days 2003*, pages 29–33. University of Kassel, 2003. ftp://ftp.upb.de/doc/techreports/Informatik/tr-ri-04-247.pdf.

[Alt07] O. Alt. Deriving reusable system test cases from SysML models. In *Proc. Software and Systems Quality Conference*, 2007.

[Ame08] C. Amelunxen. *Metamodel-based Design Rule Checking and Enforcement*. PhD thesis, Universität Darmstadt, 2008. to appear.

[BCM+94] A.W. Brown, D.J. Carney, E.J. Morris, D.B. Smith, and P.F. Zarella. *Principles of CASE Tool Integration*. Oxford University Press, 1994.

[Bec08] S. Becker. *Integratoren zur Konsistenzsicherung von Dokumenten in Entwicklungsprozessen*. PhD thesis, RWTH Aachen, 2008. German.

[Bic04] L. Bichler. *Codegeneratoren für MOF-basierte Modellierungssprachen*. PhD thesis, Universität der Bundeswehr München, 2004. German, http://www.es.tu-darmstadt.de/download/publications/bichler2004-dis.pdf.

[BNvBK06] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The Graph Rewriting and Transformation Language: GReAT. In *Proc. 3rd International Workshop on Graph Based Tools*, 2006.

[CH03] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *online Proc. 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, 2003. http://www.softmetaware.com/oopsla2003/czarnecki.pdf.

[dLV02] J. de Lara and H. Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modelling. In *Proc. 5th International Conference on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes In Computer Science (LNCS)*, pages 174–188, 2002.

[FK03] R. Freude and A. Königs. Tool integration with consistency relations and their visualization. In H. Dörr and A. Schürr, editors, *Tool-Integration in System Development Satellite Workshop of ESEC/FSE 2003, Helsinki, Finland*, pages 6–10, 2003.

[Gab02] J. Gable. Enterprise Application Integration. *Information Management Jounal*, 2002.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification*. Prentice Hall PTR, 2005.

[GK07] J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In *Model Driven Engineering Languages and Systems*, volume 4735/2007 of *LNCS*, pages 16–30. Springer Verlag, 2007.

[Gra02] M. Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, volume 1. Wiley, 2nd edition, 2002.

[Hec06] R. Heckel. Graph Transformation in a Nutshell. In *Proc. of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)*, volume 148 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 187–198, 2006.

[HM85] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. In *ACM Transactions on Information Systems*, volume 3, pages 253–278. ACM, 1985.

[JK05] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Model Transformations in Practice Satellite Workshop of MODELS 2005*, 2005.

[JKS06] J. Jakob, A. Königs, and A. Schürr. Non-materialized Model View Specification with Triple Graph Grammars. In A. Corradini, editor, *Proc. International Conference on Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science (LNCS)*, pages 321–335. Springer Verlag, 2006.

[KAK+08] F. Klar, C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. Metamodel-based Tool Integration with MOFLON. In *Proc. 30th International Conference on Software Engineering*, 2008.

[KKG05] F. Klar, T. Kühne, and M. Girschick. SPin - A Fujaba Plugin for Architecture Stratification. In *Proc. 3rd International Fujaba Days*, 2005.

[KKS07] F. Klar, A. Königs, and A. Schürr. Model Transformation in the Large. In *Proc. 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering,*, ACM Digital Library Proceedings, pages 285–294. ACM Press, 2007.

[KR02] R. Kimbal and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. Wiley, 2nd edition, 2002.

[KS06] A. Königs and A. Schürr. MDI - a Rule-Based Multi-Document and Tool Integration Approach. *Special Section on Model-based Tool Integration in Journal of Software&System Modeling*, 5(4):349–368, 2006.

[OMG02] OMG. *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, 2002. http://www.omg.org/cgi-bin/doc?ad/2002-04-10.

[OMG03] OMG. *MDA Guide Version 1.0.1*, 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[OMG05a] OMG. *MOF 2.0/XMI Mapping Specification, v2.1*, 2005. http://www.omg.org/cgi-bin/doc?formal/2005-09-01.

[OMG05b] OMG. *MOF QVT Final Adopted Specification*, 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[OMG06a] OMG. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[OMG06b] OMG. *Object Constraint Language Version 2.0*, 2006. http://www.omg.org/cgi-bin/doc?formal/2006-05-01.

[OMG07] OMG. *Unified Modeling Language: Infrastructure Version 2.1.1*, 2007. http://www.omg.org/cgi-bin/doc?formal/07-02-06.

[PR69] J. L. Pfalz and A. Rosenfeld. Web Grammars. In *Proc. 1st International Joint Conference on AI*, 1969.

[Pra71] T.W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 5:560–595, 1971. Academic Press.

[Röt04] T. Rötschke. Adding Pluggable Meta Models to FUJABA. In *Proc. 2nd International Fujaba Days*, pages 57–61, 2004.

[Röt09] T. Rötschke. *Meta Model-Based Evolution of Domain-Specific Software Architectures for Embedded Systems*. PhD thesis, TU Darmstadt, 2009. to appear.

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. PhD thesis, RWTH Aachen, 1991. German.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *Proc. WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163, Heidelberg, 1994. Springer Verlag.

[Som06] I. Sommerville. *Software Engineering*. Addison Wesley, 8th edition, 2006.

[Sun02] Sun Microsystems. *JSR-000040 The Java™ Metadata Interface (JMI) Specification Version 1.0*, 2002. http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html.

[TB94] G. Taentzer and M. Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG - an Algebraic Graph Grammar System. In H.-J. Schneider and H. Ehrig, editors, *Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science (LNCS)*, 1994.

[Wag01] R. Wagner. Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen. Diploma thesis, University of Paderborn, 2001. German.

[Wag08] R. Wagner. *Inkrementelle Modellsynchronisation*. PhD thesis, Universität Paderborn, 2008. German.

[Zha94] J.L. Zhao. Schema Coordination in Federated Database Management: A Comparison with Schema Integration. In *Proc. Workshop on Information Technology and Systems*, 1994.

[Zün01] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation thesis.